The GLAST experiment at SLAC

The Virtual Spacecraft (VSC)

GLAST Electronics group

Users Manual

Document Version:	1.4
Document Issue:	2
Document Edition:	English
Document Status:	Initial public release
Document ID:	LAT-TD-05601
Document Date:	December 5, 2005

Stanford Linear Accelerator Center (SLAC)
2575 Sandhill Road
Menlo Park California, 94025 USA

This document has been prepared using the Software Documentation Layout Templates that have been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics). For more information, go to http://framemaker.cern.ch/.

Abstract

A proposal for a SLAC produced, version of the GLAST Spacecraft (S/C), called the VSC (Virtual Spacecraft). The VSC consists of a VME crate containing modules which provide a complete implementation of the LAT to S/C ICD, including full redundancy and cross-strapping support. Within the crate resides the software necessary to control the interface to the LAT as well as software implementing an additional *external* interface. This interface, through TCP/IP sockets, allows an external application to interact with the VSC as if it were in the role of a ground-station. To do so, the application codes against a set of C++ classes called the *Proxy Interface*. A complete description and explanation of these classes is one of the principal goals of this document.

Intended audience

While, a design proposal, this document is intended principally as a guide for the *users* of the VSC. These include:

- Testers of Flight Software
- Developers of Flight Software
- Developers of I&T (Integration and Test) based systems

Conventions used in this document

Certain special typographical conventions are used in this document. They are documented here for the convenience of the reader:

- Field names are shown in bold and italics (*e.g.*, ***respond*** or ***parity***).
- Acronyms are shown in small caps (*e.g.*, SLAC or TEM).
- Hardware signal or register names are shown in Courier bold (*e.g.*, `RIGHT_FIRST` or `LAYER_MASK_1`)

References

1 *1553 Product Handbook*, United Technologies Microelectronics Center Inc., 1992.

2 *CCSDS Package User Manual,* LAT Flight Software User Manual.

3 *CTDB 1553 Drivers,* LAT Flight Software User Manual.

4 *EGSE Used to Test GASU*, LAT-TD05787.

5 *Enhanced Summit Family Product Handbook*, UTMC Microelectronics Systems Inc., October 1999.

6 *GLAST 1553 Bus Protocol Interface Control Document*, Spectrum Astro, Inc.

7 *GLAST LAT Instrument – Spacecraft Interface Requirements Document, 433-IRD-0001 Revision B*, NASA Goddard Space Flight Center, April 2002.

8 *GLAST LAT to Spacecraft Interface Control Document*, Spectrum Astro, Inc., February 2003.

9 *GLAST Spacecraft Interface Board Hardware Specification*, LAT Hardware Specification Document.

10 *LTX User Manual*, V1-1-0, LAT FSW User Manual, August 2003.

11 *Military Standard 1553B Notice 2*, United States Department of Defence, September 1986.

12 *Military Standard 1553B, Aircraft Internal Time Division Command/Response Multiplex Data Bus*, United States Department of Defence, September 1978.

13 *MSG User Manual*, LAT Flight Software User Manual.

14 *PBS Package Documentation,* LAT Flight Software Code Documentation.

15 *PMC-1553 Reference Manual, Rev 1.2*, Alphi Technology Corporation, June 1998.

16 *Recommendation for Space Data System Standards, Advanced Orbiting Systems, Networks and Data Links: Architectural Specification, Blue Book 701.0-B-3*, Consultative Committee for Space Data Systems, June 2001.

17 *Recommendation for Space Data System Standards, Telecommand Part 3, Data Management Service Architectural Specification, Blue Book 203.0-B-1*, Consultative Committee for Space Data Systems, January 1987.

18 *Telecommand and Telemetry Formats*, LAT-TD-02659.

19 "GASU Based Teststands - A hardware and software Primer," Michael Huffer, LAT-TD-03664.

20 *Symmetricom* TTM635/637VME Time and Frequency Processor, revision B Users Guide, 8500-0138 February, 2004

21 LAT Flight Software, CMX Manual, version V2-2-3, *A. P. Waite*, updated 30, November 2004

22 "The GLT Electronics Module- Programming ICD specification," Michael Huffer, LAT-TD-01545.

23 See the LAT Flight Software, web-site for the traveller documentation for LPA (currently undefined)

24 See the LAT Flight Software, web-site for the traveller documentation for datagrams (currently undefined)

25 See the LAT Flight Software, web-site for the traveller documentation for LCI (currently under project APP, package LCI)

26 "The EBM Electronics Module- Programming ICD specification," Michael Huffer, LAT-TD-01546.

27 "The DataFlow Public Interface," Michael Huffer, LAT-TD-01546.

28 "Data Sheet for the Goodrich Model 0118MF High Reliability Surface Temperature Sensor," http://www.sensors.goodrich.com/literature/list.shtml

29 "Data Sheet for the YSI 44900 Series Thermistor (GSFC S-311-P-18)", http://www.ysitemperature.com/tech-home.html

Note: For additional resources, refer to the LAT Electronics, DAQ Critical Design Requirements List. On the LAT Electronics, Data Acquisition & Instrument Flight Software page (http://www-glast.slac.stanford.edu/Elec_DAQ/Elec_DAQ_home.htm), click Hardware and then click List of all documents.

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	The Virtual Spacecraft (VSC) Users Manual		
	Version:	1.4		
	Issue:	2		
	Edition:	English		
	ID:	LAT-TD-05601		
	Status:	Initial public release		
	Created:	February 9, 2002		
	Date:	December 5, 2005		
	Access:	V:\GLAST\Electronics\Design Documents\VSC\V1.4\frontmatter.fm		
	Keywords:	GASU Based Teststands		
Tools	**DTP System:**	Adobe FrameMaker	**Version:**	6.0
	Layout Template:	Software Documentation Layout Templates	**Version:**	V2.0 - 5 July 1999
	Content Template:	--	**Version:**	--
Authorship	**Coordinator:**	Michael Huffer		
	Written by:	Michael Huffer		

Table 2 Approval sheet

Name	Title	Signature	Date
Gunther Haller	LAT CHIEF ELECTRONICS ENGINEER		
JJ Russell	FLIGHT SOFTWARE LEAD		

Document Status Sheet

Table 3 Document Status Sheet

Title:	The Virtual Spacecraft (VSC) Users Manual		
ID:	LAT-TD-05601		
Version	**Issue**	**Date**	**Reason for change**
1.0	1	8/22/2005	First public release
1.2	1	11/10/2005	Added support for the assembly of science packets to datagrams. See Section 4.8 and Appendix A.
1.3	1	11/21/2005	Added support for the LAT voltage and temperature monitoring. See Section 3.3.6, Section 7.10, and Appendix B.
1.4	1	12/05/2005	After some reflection, changed structure of monitoring packets. Added support for conversion from ADC counts to engineering units for these packets. See Appendix B.

Table of Contents

Abstract 3

Intended audience 3

Conventions used in this document 3

References 4

Document Control Sheet 6

Document Status Sheet 7

List of Tables 15

List of Figures 17

List of Listings 19

Chapter 1
Introduction 21

1.1 Information Exchange 22

1.1.1 Information exchange between LAT and VSC 23

1.1.2 Information Exchange between VSC and proxy interface 24

1.2 Software methodology and organization 25

1.2.1 Documentation conventions 27

1.2.2 Header files and name spaces 28

1.2.3 Configuration management 28

1.3 Observatory time 28

1.3.1 Time representation 29

1.3.1.1 Initializing the observatory time-base 29

1.3.2 Time Keeping 30

1.4 Routing and scheduling requests 30
1.4.1 Routing 31
1.4.2 Queuing 32
1.4.3 Scheduling 33
1.4.3.1 Harvesting the work load 35
1.4.3.2 Scheduling state transitions 35
1.4.3.3 Scheduling control work 35
1.4.3.4 Scheduling command work 36
1.4.3.5 Cleanup 36
1.4.4 Scheduler State Model 37

Chapter 2
Hardware 41
2.1 Electrical conventions 42
2.2 The SBC and 1553 interface 42
2.3 The GPS receiver 43
2.4 The Science interface 44
2.5 The Discrete interface 45
2.6 Digitizer 46
2.7 LAT Cable interface 46
2.8 LAT Monitoring interface 47
2.9 VSC Configurations 48
2.9.1 The Testbed 48
2.10 VSC Inter-Connectivity 51

Chapter 3
Using the proxy interface 53
3.1 Distribution of telemetry 55
3.2 Managing the SC/GBM interface 57
3.3 Managing the SC/LAT interfaces 58
3.3.1 SIU cross-strapping 58
3.3.2 SIU Reset 59
3.3.3 SIU discretes 60
3.3.4 GASU (DAQ board) cross-strapping 60
3.3.5 Enabling the Science Interface 62
3.3.6 Enabling Diagnostic Monitoring 62
3.4 Handling Incoming telemetry 64
3.4.1 APID filtering 65
3.5 Routing Incoming telemetry 66
3.5.1 Router registration 68
3.5.2 APID dispatching 69
3.5.3 Science telemetry and datagrams 70
3.6 Telecommands 70

3.7 Scheduling the “Magic seven” . . . 71
3.8 Administrating the VSC . . . 74
3.8.1 Configuring queue sizes . . . 74
3.8.2 Network configuration . . . 75
3.8.2.1 VSC node name and IP address . . . 75
3.8.2.2 Port Numbers . . . 76

Chapter 4
CCSDS package . . . 77
4.1 Name space - `VscCcsds` . . . 78
4.2 Packet sequencing . . . 78
4.3 Packet . . . 78
4.3.1 Constructor synopsis . . . 79
4.3.2 Member synopsis . . . 80
4.4 TeleCmnd . . . 80
4.4.1 Constructor synopsis . . . 81
4.4.2 Member synopsis . . . 82
4.5 Mangle . . . 82
4.5.1 Constructor synopsis . . . 83
4.5.2 Member synopsis . . . 83
4.6 Generic Telemetry packet . . . 84
4.6.1 Constructor synopsis . . . 85
4.6.2 Member synopsis . . . 86
4.7 Telemetry . . . 86
4.8 Science . . . 87
4.8.1 Member synopsis . . . 87
4.9 The “Magic 7” Telecommands . . . 88
4.9.1 Constructor synopsis . . . 88
4.9.2 Member synopsis . . . 88
4.10 The Attitude Ancillary Telecommand . . . 89
4.10.1 Constructor synopsis . . . 89
4.10.2 Member synopsis . . . 90
4.11 The Ancillary Data Telecommand . . . 90
4.11.1 Constructor synopsis . . . 92
4.11.2 Member synopsis . . . 92
4.12 The Time-Tone Ancillary Telecommand . . . 93
4.12.1 Constructor synopsis . . . 94
4.12.2 Member synopsis . . . 94
4.13 Exceptions . . . 94

Chapter 5

The Handling package 95
5.1 Name space - `VscHandling` 96
5.2 Handler 96
5.2.1 Constructor synopsis 97
5.2.2 Member synopsis 97
5.3 Telemetry Handler 98
5.3.1 Constructor synopsis 98
5.3.2 Member synopsis 98
5.4 Science Handler 98
5.4.1 Constructor synopsis 99
5.4.2 Member synopsis 99
5.5 Telecommand Handler 99
5.5.1 Constructor synopsis 99
5.5.2 Member synopsis 99
5.6 APID Range 99
5.6.1 Constructor synopsis 100
5.6.2 Member synopsis 100
5.7 Exceptions 101

Chapter 6
The Routing package 103
6.1 Name space - `VscRouting` 104
6.2 Router 104
6.2.1 Constructor synopsis 105
6.2.2 Member synopsis 105
6.3 Telemetry Router 106
6.3.1 Constructor synopsis 106
6.3.2 Member synopsis 106
6.4 Science Router 106
6.4.1 Constructor synopsis 107
6.4.2 Member synopsis 107
6.5 Telecommand Router 107
6.5.1 Constructor synopsis 107
6.5.2 Member synopsis 107
6.6 Exceptions 108
6.6.1 Insufficient Memory 108

Chapter 7
The VSC proxy package 109
7.1 Name space - `VscProxy` 110
7.2 Proxy 110
7.2.1 Constructor synopsis 112
7.2.2 Member synopsis 112

7.3 Scheduler control request 116
7.3.1 Constructor synopsis 116
7.3.2 Member synopsis 116
7.4 Control request 117
7.5 Cross-strapping options 117
7.6 SIU interface control request 118
7.6.1 Constructor synopsis 118
7.6.2 Member synopsis 118
7.7 DAQ interface control request 118
7.7.1 Constructor synopsis 119
7.7.2 Member synopsis 119
7.8 SIU discrete control request 119
7.8.1 Constructor synopsis 120
7.8.2 Member synopsis 120
7.9 SIU reset control request 121
7.9.1 Constructor synopsis 121
7.9.2 Member synopsis 121
7.10 Monitor control request 121
7.10.1 Constructor synopsis 122
7.10.2 Member synopsis 122
7.11 SSR control request 122
7.11.1 Constructor synopsis 123
7.11.2 Member synopsis 123
7.12 Down load control request 123
7.12.1 Constructor synopsis 123
7.12.2 Member synopsis 123
7.13 GBM control request 124
7.13.1 Constructor synopsis 124
7.13.2 Member synopsis 124
7.14 Proxy parameters 124
7.14.1 Constructor synopsis 124
7.14.2 Member synopsis 125
7.15 Exceptions 125
7.15.1 Data Stream Allocated 125
7.15.2 No transmit port 125
7.15.3 Network Transmit Failure 125

Appendix A
The Datagram support package 127
A.1 Name space - `VscDatagram` 128
A.2 Datagram Assembler 128

A.2.1 Constructor synopsis . . . 129
A.2.2 Member synopsis . . . 129
A.3 LCI Assembler . . . 130
A.3.3 Constructor synopsis . . . 130
A.3.4 Member synopsis . . . 130
A.4 LPA Assembler . . . 130
A.4.5 Constructor synopsis . . . 131
A.4.6 Member synopsis . . . 131
A.5 Exceptions . . . 131

Appendix B
Telemetry from the Monitoring System . . . 133
B.1 Unit conversion . . . 133
B.1.1 LAT voltage (LatV) . . . 134
B.1.2 BPU voltage (BpuV) . . . 134
B.1.3 BPU Current (BpuI) . . . 134
B.1.4 DAQ Current (DaqI) . . . 134
B.1.5 Thermistor . . . 135
B.1.6 RTD . . . 135
B.2 Telemetry from the 850 board . . . 135
B.2.7 APID = `0x00A8` . . . 137
B.3 Telemetry from the 468 board . . . 138
B.3.8 APID = `0x00A4` . . . 139

List of Tables

Table 1 p. 6 Document Control Sheet
Table 2 p. 6 Approval sheet
Table 3 p. 7 Document Status Sheet
Table 4 p. 33 Queue scheduling order
Table 5 p. 59 Cross-strapping options for the SIU interface
Table 6 p. 61 Cross-strapping options for the DAQ interface
Table 7 p. 68 Streams and registration methods
Table 8 p. 78 Enumeration of sequence flag for a CCSDS telemetry packet
Table 9 p. 117 Cross-strapping options

List of Figures

Figure 1 p. 22 Logical relationship of the VSC and its proxy interface to the observatory
Figure 2 p. 23 Logical relationship of the VSC and its external interface to the observatory
Figure 3 p. 26 Class package inter-dependencies
Figure 4 p. 31 Command and Control flow through the VSC
Figure 5 p. 32 VSC queue, both immediate and stored.
Figure 6 p. 34 Scheduling within a period
Figure 7 p. 34 Phases within a period
Figure 8 p. 38 VSC Finite State Machine Diagram
Figure 9 p. 42 SBC and 1553 interface
Figure 10 p. 43 GPS receiver
Figure 11 p. 44 Science interface
Figure 12 p. 45 Vsc components
Figure 13 p. 46 Digitizer
Figure 14 p. 47 VSC-850 module
Figure 15 p. 49 Corner teststand configuration
Figure 16 p. 50 Internal Connectivity (LAT VSC)
Figure 17 p. 51 External Connectivity (LAT VSC)
Figure 18 p. 56 One proxy interface processing all telemetry streams
Figure 19 p. 56 Multiple proxy interfaces
Figure 20 p. 58 SIU cross-strapping
Figure 21 p. 61 DAQ board cross-strapping
Figure 22 p. 72 Scheduling the “Magic seven”
Figure 23 p. 77 Class dependencies for the CCSDS package

Figure 24 p. 95 Class dependencies for the Handler package
Figure 25 p. 103 Class dependencies for the Routing package
Figure 26 p. 110 Class dependencies for the proxy package
Figure A.1 p. 127 Class dependencies for the datagram support package
Figure B.1 p. 137 Enumeration of 850 board monitored quantities
Figure B.2 p. 139 Enumeration of 468 board monitored quantities

List of Listings

Listing 1 p. 78 Enumeration for packet sequence flags
Listing 2 p. 79 Class definition for `Packet`
Listing 3 p. 81 Class definition for `TeleCmnd`
Listing 4 p. 83 Class definition for `Mangle`
Listing 5 p. 85 Class definition for `GenericTelemetry`
Listing 6 p. 87 Class definition for `Telemetry`
Listing 7 p. 87 Class definition for `Science`
Listing 8 p. 88 Class definition for `M7Cmnd`
Listing 9 p. 89 Class definition for `Attitude`
Listing 10 p. 91 Class definition for `Data`
Listing 11 p. 93 Class definition for `TimeTone`
Listing 12 p. 96 Class definition for `Handler`
Listing 13 p. 98 Class definition for `TelemetryHandler`
Listing 14 p. 98 Class definition for `ScienceHandler`
Listing 15 p. 99 Class definition for `TeleCmndHandler`
Listing 16 p. 100 Class definition for `ApidRange`
Listing 17 p. 105 Class definition for `Router`
Listing 18 p. 106 Class definition for `TelemetryRouter`
Listing 19 p. 107 Class definition for `ScienceRouter`
Listing 20 p. 107 Class definition for `TeleCmndRouter`
Listing 21 p. 111 Class definition for `Proxy`
Listing 22 p. 116 Class definition for `Scheduler`
Listing 23 p. 117 Class definition for `Control`
Listing 24 p. 118 Class definition for `SiuInterface`

Listing 25 p. 119 Class definition for `DaqInterface`
Listing 26 p. 120 Class definition for `ToggleLines`
Listing 27 p. 121 Class definition for `Reset`
Listing 28 p. 122 Class definition for `Monitor`
Listing 29 p. 122 Class definition for `SsrInterface`
Listing 30 p. 123 Class definition for Download
Listing 31 p. 124 Class definition for`Grb`
Listing 32 p. 124 Class definition for `Parameters`
Listing A.1 p. 128 Class definition for `Assembler`
Listing A.2 p. 130 Class definition for `LciAssembler`
Listing A.3 p. 131 Class definition for `LpaAssembler`

Chapter 1
Introduction

Physically, the VSC consists of a VME crate with a Single-board Computer (SBC) and up to five different modules[1]. The processor executes code operating under *VxWorks* as its real-time kernel. As its principal responsibility, the VSC emulates the spacecraft's (S/C) side of the interface between spacecraft and LAT[2]. A series VME modules implement the physical interface. These modules are controlled and managed with software residing in the SBC. This software constitutes the VSC's *internal* implementation which is not described in this document. In addition to implementing the ICD, the VSC also includes an *external* software interface. The external interface allows an application to control and manage the VSC *remotely.* The remote interface is called the VSC*'s Proxy Interface*. This interface resides on a remote platform (both UNIX and *Windows* based platforms). The sum total of the VSC's hardware, internal software, and external interface constitute a spacecraft simulation. Other simulations of the spacecraft exist, each with complementary features. However, the VSC is designed expressly to meet the specific requirements of the LAT and in particular address the needs of the FSW group with respect to the simulation of the science data acquired by the LAT. To this end, the VSC has some features *not* required of the spacecraft:

- interface to the FES (Front-End Simulator)
- the ability to set the observatory time to an arbitrary value
- the ability to start and stop observatory time
- the ability to drive, externally, the VSC's absolute time and position model
- "real-time" access to science data

Any application, coded against the proxy interface, should partition these features in such a way that as the VSC is moved into a more realistic observatory environment, these features can either appropriately mutate or be disabled.

An application coded against the proxy interface would provide its users the fiction of operating the LAT as if it were on-orbit. Logically, if the VSC represents a simulation of the

1. A description of the VSC hardware is found in Chapter 2.
2. Hereafter, referred to as the Spacecraft (S/C) ICD (Interface Control Document).

spacecraft, then the application, coded against the proxy interface, would represent a simulation of the ground station. Because the proxy interface is a *programmatic* interface, a variety of ground station simulations could be envisioned. For example, "ground stations" whose client is:

- FSW and FSW Test, in order to both develop and test the LAT's flight software system
- I & T ("LATTE 5"), in order to manage the integration and test of the instrument
- The ISOC, in order to both develop and test the instrument's ground station

The relationship between the role of the VSC, its proxy interface, the Spacecraft, LAT, and ground station is illustrated in Figure 1:

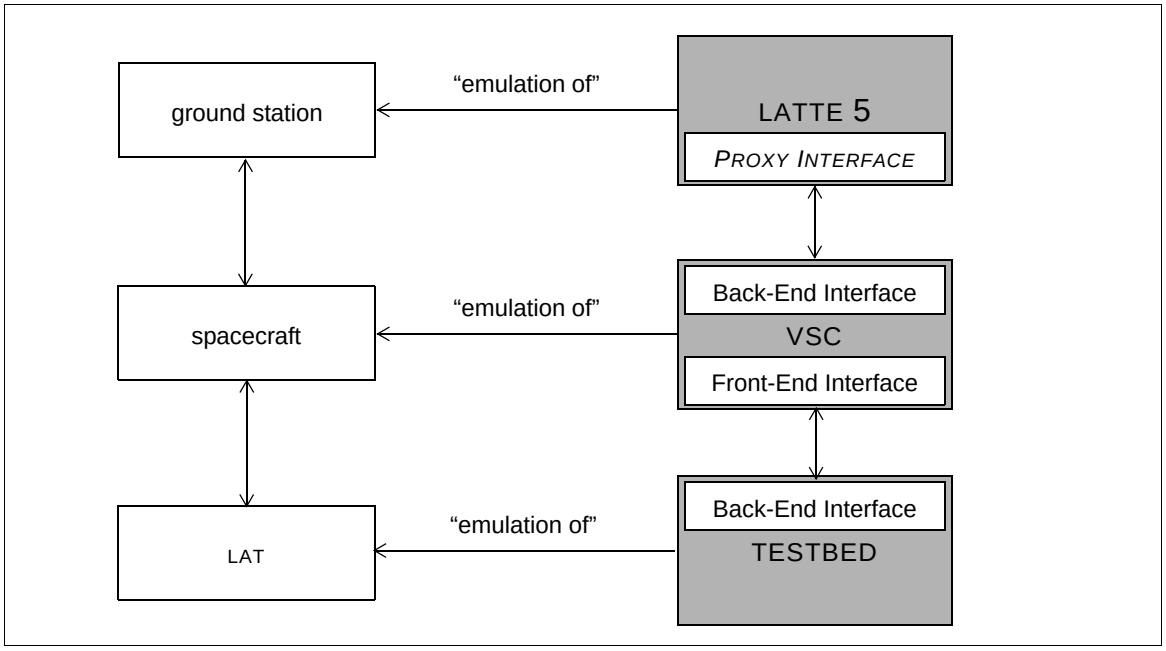

Figure 1 Logical relationship of the VSC and its proxy interface to the observatory

Where here (as one example) the emulation of the ground station is "LATTE 5". Recall the LAT itself represents an abstraction. This abstraction is realized in many forms, three examples are: the test-stand, the test-bed, and of course, the physical realization of the real LAT. Therefore any application implementing the proxy interface may inter-operate (through the VSC) with any representation of the LAT, whether it's the "real thing", the testbed, or a teststand. This provides the user with a powerful tool to debug and commission their own applications. As an example, within Figure 1, the emulation of the LAT is the *testbed* residing in the DataFlow lab.

1.1 Information Exchange

Vertical arrows shown between entities in Figure 1 represents flow of information. Information is exchanged between entities using a variety of protocols, with the protocol

varying on both the type of information exchanged and the information's latency and bandwidth requirements. Therefore, the information flow represented by a single arrow in Figure 1 can be further decomposed into a set of individual data *streams* as illustrated in Figure 2:

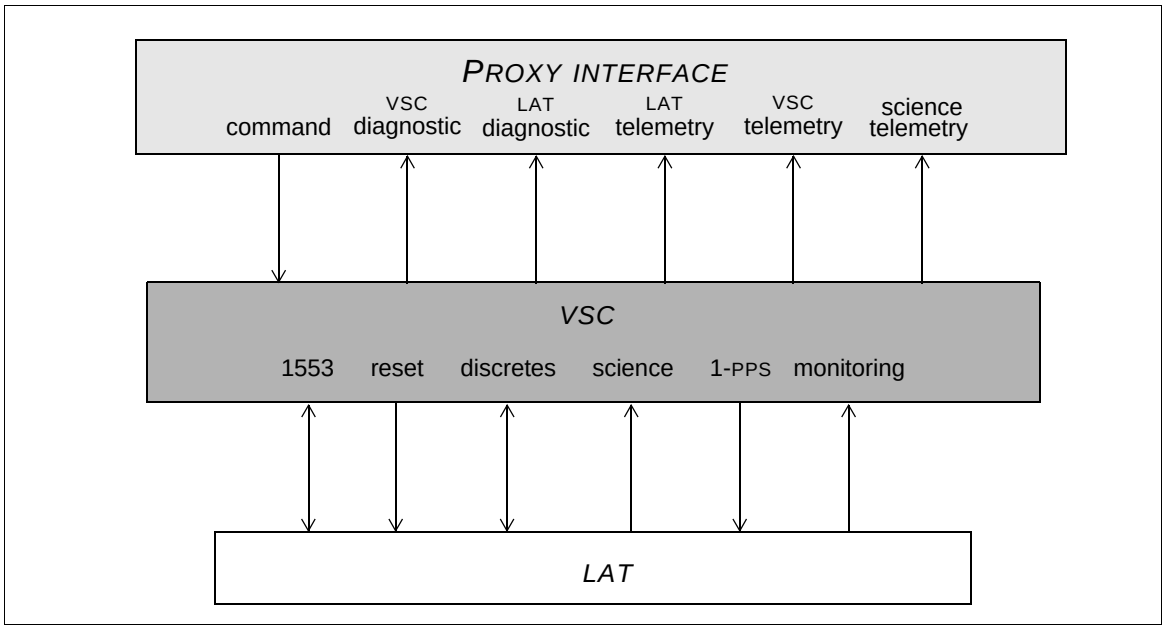

Figure 2 Logical relationship of the VSC and its external interface to the observatory

Each arrow represents not only the type of information exchanged on the stream, but also that information on any one stream is exchanged asynchronously with respect to information exchange on any other stream. The direction of the arrow represents the direction information flows on the stream. Note that for some streams information flow is bi-directional.

1.1.1 Information exchange between LAT and VSC

1553: All information transferred by the 1553 stream is encapsulated as CCSDS packets and these packets can flow in either direction between VSC and LAT. The LAT and VSC specify the type of these packets as either *telecommands* or *telemetry* (see chapter 4). Telecommands received by a recipient direct that recipient to perform some generic type of action. Telemetry received by a recipient contains generic information about the state or health of the entity sending the packet. The LAT breaks its telemetry arbitrarily into two types: *diagnostic* and *housekeeping* (see [18]). An example of a telecommand would be a dump memory request sent by the VSC to the LAT specifying an upload of a specific piece of memory on the processor within a SIU. An example of telemetry would be a packet sent by the LAT to the VSC describing the current bias voltage of an ACD FREE board.

Telecommands can either be directed to the LAT by the VSC, or directed to the VSC by the LAT. Telemetry may be received only by the VSC from the LAT. The set of possible telecommand and telemetry packets exchanged between LAT and VSC is defined by the observatory *Command and Telemetry Database* (see [18]).

reset: Is a specific type of discrete request sent by the VSC to the LAT. This request issues a reset to the currently selected SIU. This request causes not only a re-boot of the SIU's processor, but also causes the SIU to issue a global reset to the entire LAT.

discretes: Discretes are simply digital (boolean) signals exchanged between LAT and VSC. The VSC can assert/deassert *three* (3) different discrete signals to the currently selected SIU. The currently selected SIU can assert/deassert *four* (4) different signals to the VSC. These signals are typically used as boot flags between VSC and LAT (see [8]).

science: Information sent on this stream contains the LAT's science data. This data includes both the LAT's accepted events and housekeeping (science telemetry). Independent of type of data, information sent on this stream is encapsulated as CCSDS packets.

1-PPS: This is a signal (pulse) sent by the VSC to the LAT once a second. This signal is sent with great precision by the VSC to allow accurate correlation of time between VSC and LAT. The absolute time corresponding to any one particular pulse is sent sometime later by the VSC to the LAT as a 1553 telecommand (the time-tone message, see Section 4.12). Normally this pulse and its corresponding absolute time is maintained with great precision by the VSC using a GPS receiver. See Section 1.3 for a discussion of how time is maintained on the VSC.

monitoring: This is a predefined set of analog voltage and temperatures on the LAT, but measured by the VSC. The VSC uses these measurements as input into the decision to power-on and boot the LAT. The measurement of these temperatures and voltage are *not* part of the LAT's housekeeping telemetry as they must be monitored even when the LAT is not operating. Instead, this information is sent as VSC telemetry (see appendix B).

1.1.2 Information Exchange between VSC and proxy interface

command: This stream transmits CCSDS telecommands from the proxy interface to the VSC. Telecommands fit into two classes: *Control requests* and *Commands*. Control Requests are telecommands consumed entirely within the VSC and generally constitute the functions to manage and control the VSC and its interfaces to the LAT. One example of a request would be the telecommand used to establish which of two SIU's (primary or redundant) the VSC communicates with on the LAT. The set of telecommands which are allowed on this stream are enumerated and described in xxx. Commands are telecommands whose final destination is the LAT. These commands are queued on the VSC for later delivery to the LAT. The set of commands allowed on this stream is enumerated and described within the "The GLAST Command and Telemetry Database" (see [18]).

VSC diagnostic: This stream transmits *solicited* CCSDS telemetry from the VSC to the proxy interface. Solicited telemetry is requested by the user's of the proxy interface, by queuing *download* requests (see Section 7.12). The set of CCSDS packets which constitute the allowed telemetry on this stream are enumerated and described in appendix B. Note that this stream is independent of the ***VSC telemetry*** stream used to receive unsolicited telemetry sourced by the VSC. That stream is the ***VSC telemetry*** stream described below.

LAT diagnostic: This stream transmits CCSDS telemetry from the VSC to the proxy interface. This telemetry is the LAT diagnostic telemetry which was brought into the VSC through the ***1553*** interface between VSC and LAT. The set of CCSDS packets which constitute the allowed telemetry on this stream are enumerated and described within the "The GLAST Command and Telemetry Database" (see [18]). Note that this stream is independent of the ***LAT telemetry*** stream. That stream is the ***LAT telemetry*** stream described below.

VSC telemetry: This stream transmits un*solicited* CCSDS telemetry from the VSC to the proxy interface. An example of such telemetry would be the voltage and temperatures of the LAT which are measured by the VSC. The set of CCSDS packets which constitute the allowed telemetry on this stream are enumerated and described in xxx. Note that this stream is independent of the ***LAT telemetry*** stream used to receive housekeeping and diagnostic telemetry sourced by the LAT, but which are brought out to the proxy interface *through* the VSC. That stream is the ***LAT telemetry*** stream described below.

LAT telemetry: This stream transmits CCSDS telemetry from the VSC to the proxy interface. This telemetry is the LAT housekeeping and diagnostic telemetry which was brought into the VSC through the ***1553*** interface between VSC and LAT. The set of CCSDS packets which constitute the allowed telemetry on this stream are enumerated and described within the "The GLAST Command and Telemetry Database" (see [18]). Note that this stream is independent of the ***VSC telemetry*** stream. That stream is the ***VSC telemetry*** stream described above.

Science: This stream transmits CCSDS telemetry from the VSC to the proxy interface. This telemetry is the LAT *science* telemetry which was brought into the VSC through the ***science*** interface between VSC and LAT. The ***science*** stream carries both the LAT's events and science telemetry. The set of CCSDS packets which constitute the allowed telemetry on this stream are enumerated and described within the "The GLAST Command and Telemetry Database" (see [18]).

1.2 Software methodology and organization

Software used for both the VSC and its proxy interface is based on object-oriented model, with C++ as the implementation language. Consequently, the interface is expressed primarily as a series of *classes*. Many of the classes work together to provide a single coherent set of services. These classes are grouped together to form a *class package*[1]. The VSC software is comprised of

ten of these packages. Each package corresponds to both an individual name space and individual sets of header files (see Section 1.2.2). The relationship between packages is expressed in Figure 3:

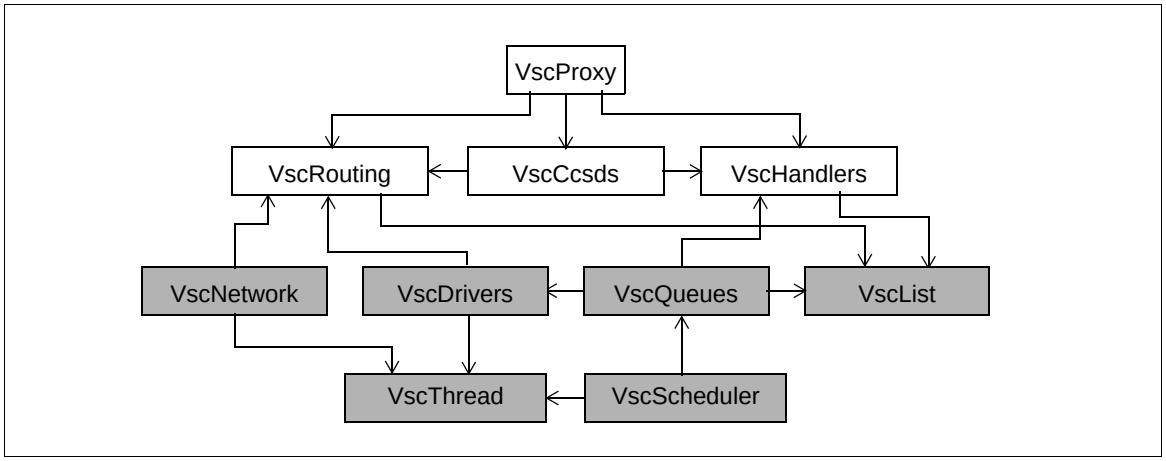

Figure 3 Class package inter-dependencies

Each box represents a single package. The name of the package is contained within the box. Arrows between boxes represent the dependencies between packages, with the direction of arrow pointing *to* the dependent package. For example, `VscRouting` and `VscDrivers` are both dependent on `VscDataHanders`. Dependencies can be of three different forms:

- inheritance
- "has a" relationship
- "uses" relationship

Shaded boxes represent the packages used exclusively for implementation and are expected to have no direct interest to a user or their applications. The four unshaded boxes constitute the proxy interface and contain those classes which define the public (proxy) interface. It is from these three packages that a user's application will be constructed. A summary of the function of these packages would be:

VscCcsds: The quanta of information exchange between VSC and proxy interface is a CCSDS packet (see [16] and [17]). The observatory differentiates between two types of CCSDS packets: *telecommands* and *telemetry*. Telecommands are *sent* by the proxy interface to direct either the VSC or LAT to perform some generic type of action. Telemetry is *received* from the VSC by the proxy interface[1] and contains generic information about the state or health of either the VSC or LAT. The CCSDS packet package contains a variety of classes to both instantiate and inspect CCSDS packets representing either telecommands or telemetry. See Chapter 4 for a detailed description of the classes of this package.

1. Unfortunately, the usage of the word *package* in this context is quite distinct from its usage by CMX (see Section 1.2.3).

1. Although always received from the VSC, telemetry may originate from either the VSC or the LAT.

VscHandlers: This package provides the mechanisms to "catch" any arrived telemetry. Telemetry arrives on three different streams: the ***VSC telemetry, LAT telemetry, Science telemetry*** streams. The application provides an individual handler for each stream. This handler is used to catch and processes telemetry packet arriving on that particular stream. Each stream is associated with a separate *thread* (see Section 3.5). It is in the context of that thread that the application's handler executes. Chapter 5 contains a detailed description of the classes of this package.

VscProxy: This package forms the anchor around which the application's implementation is built. The package provides the following services:

- Connects to the VSC commanding stream.
- Allows an application to connect to any one of the five telemetry streams. When connecting, the application registers a handler to catch and process any and all telemetry packets arriving on the connected stream.
- Allows the user to *queue* on the VSC any LAT telecommand for subsequent delivery and execution on the LAT.
- Allows the user to *queue* on the VSC any VSC telecommand for latter delivery and execution on the VSC. These telecommands' construction and transmission are hidden behind another set of eight classes. Each class groups together related telecommands into a single coherent interface. The functions of these classes are to:
  - control the VSC's scheduler
  - select and monitor SIU cross-strapping
  - select and monitor DAQ board cross-strapping on the GASU
  - control and monitor the FES (Front-End Simulator)
  - queue a GRB (Gamma Ray Burst) request to the LAT
  - monitor and control the state of the "device ready" line of the science interface between VSC and LAT
  - monitor and control the discrete interface between VSC and LAT
  - define the scheduling of the seven different ancillary telecommands for each and every second of the VSC's operation

  See Chapter 7 for a detailed description of the classes of this package.

Each package is assigned its own individual chapter. These chapter provide a detailed description of the classes contained within the package. For example, the `VscCcsds` package is described in chapter 4.

1.2.1 Documentation conventions

Each package is described individually in its appropriate chapter. Each description starts with a synopsis of the package's function, followed by a class dependency diagram expressing the

relationships between the classes of the package and the package's linkages to other classes in other packages. The conventions for these diagrams are as follows:

- Each box represents a class.
- A shaded box indicates a class which is outside the package and on which at least one of the package's classes is dependent.
- Arrows specify a relationship between classes. A solid line specifies an "inheritance" relationship. A dotted line specifies either a "uses" or a "has a" relationship.
- The direction of the arrow is such that an arrow points *away* to the dependent class.

Following the class diagram is a description for each of the classes of the package. Each class description begins with a class specification. The specification syntax is C++ as this is implementation language for both the VSC and its proxy interface. Following the specification is a short synopsis of the class's constructors and members. The package description ends with an enumeration of the exceptions the classes of the package may throw.

1.2.2 Header files and name spaces

Each package has its own name space. The name of this space corresponds to the name of the package. Each class within the package has two associated files:

- A file with an extension "`.hh`" containing the interface (class) description
- A file with an extension of "`.cc`" containing the class's implementation

Each file name has the form: `PackageName-ClassName`. For example, the class description file of the CCSDS telemetry packet is a part of the `VscCCSDS` package and would, therefore, have the file name "`VscCcsds-Telemetry.hh`", while its corresponding implementation file would have the name "`VscCcsds-Telemetry.cc`".

1.2.3 Configuration management

The code base is managed using the FSW (Flight-Software) management and build system (see [21]). The code base is organized under its own project. The project name is *VSC*.

The VSC package and constituent organization remains to be described.

1.3 Observatory time

The VSC has the responsibility for maintaining consistent time between it and the LAT. In turn, the LAT is responsible for insuring its clock is consistent both within itself and the FES (Front-End-Simulator)[1]. In this fashion, time is coherently maintained between spacecraft, instrument, and event simulation. The LAT insures consistency between it and the FES by

driving the FES's clock using the LAT external test connector (see [19]). Coherency between VSC and LAT is a bit more complex and requires understanding how the VSC represents and keeps time.

1.3.1 Time representation

The VSC represents time as an unsigned 32-bit value, in which one *count* equals one *second*. A value of *zero* corresponds to the time at 00:00:00.0 hours at January 1st, 2001. This is the common unit of time measurement between spacecraft and instrument (see [8]). The agreed upon *current* time between VSC and LAT is maintained within the VSC's *observatory time-base*. The time-base is updated once a second (see Section 1.3.2), therefore, its value at any one point corresponds to the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The VSC uses the observatory time-base to:

— determine when a telecommand becomes due (Section 1.3)

— Set the period time for all ancillary commands

— time-tag any telemetry the VSC might source

— time-tag any necessary timing information sent to the LAT

In short, wherever the VSC requires an absolute time, it does so by accessing the observatory time-base.

1.3.1.1 Initializing the observatory time-base

In addition to the observatory time-base the VSC also maintains a local or *wall-clock* time-base. Both time-bases are initilized whenever the VSC is booted. First the wall-clock time-base is initialized. Its initial value depends on the presence or absence of a GPS receiver within the VSC (see Chapter 2).

— If a GPS receiver is present, the time-base is set to the current universal time as determined by the GPS receiver.

— If a GPS receiver is *not* present, the time-base is set to the current universal time as determined by the local *vxWorks* real-time clock (see xxx).

Once the wall-clock time -base is established, the observatory time-base is simply initialized to the current value of the wall-clock time-base. However, at any time, the base value of the observatory time-base can also established via a request through the proxy interface (see Section 7.4). The updated initial value depends on whether or not the request specifies a a particular value:

— If a request specifies a particular time, the time-base will be set to that time.

1. When the LAT is implemented within the testbed.

— If a request does not specify a particular time (the "don't care" value), the time-base will be set to the current value of the wall-clock time-base.

Note, that whatever the mechanisms the time-base is established (booting or requested) the the VSC scheduler must be in the *stopped* state (see Section 1.4.3). Setting an explicit time allows the VSC's time to be consistent with, for example, a physics simulation which drives the LAT through the FES.

1.3.2 Time Keeping

Both the VSC and LAT are slaved to a hardware generated 1-PPS (One-Pulse Per Second) signal. The presence or absence of a GPS receiver (see Chapter 2), determines which one of two types of 1-PPS signal is used:

— A *stable* source. A 1-PPS signal derived by VME based GPS receiver. In turn, The GPS receiver may derive its timing externally (from satellite), or internally ("fly wheel").

— A *degraded* source. A 1-PPS signal derived from a clock driven by the VSC's science interface (see Chapter 2).

On boot, the VSC chooses for its 1-PPS signal the most stable source available. The 1-PPS signal has four functions:

i. re-synchronizes the wall-clock

ii. increments the observatory time-base

iii. triggers the VSC's scheduler (see Section 1.4)

iv. is transmitted to the LAT

Shortly before the 1-PPS signal is transmitted to the LAT, the VSC generates the so-called "time-tone" telecommand (see Section 4.12). This command announces the observatory time for corresponding 1-PPS signal. This telecommand is one of the seven ancillary commands sent once a second to the LAT (see [8]).

Note, that one field of this command specifies whether or not the source of the 1-PPS signal was a GPS receiver. The value of this field is independent of which of the two types of sources are used by the VSC.

1.4 Routing and scheduling requests

The command and control of the VSC is governed by five different types of objects:

inPorts: Delivers request packets off a stream.

routers: Routs arrived requests, based on APID, to their appropriate queue.

queues: Holds telecommands in time-order for their later execution.

scheduler: Examines, once a second, each queue to determine whether there are any telecommands due for execution. Scheduling is discussed in Section 1.4.3.

drivers: Transforms telecommands to appropriate LAT interface commands. Sends these commands to the LAT.

The flow of command and control through these objects is illustrated within Figure 4:

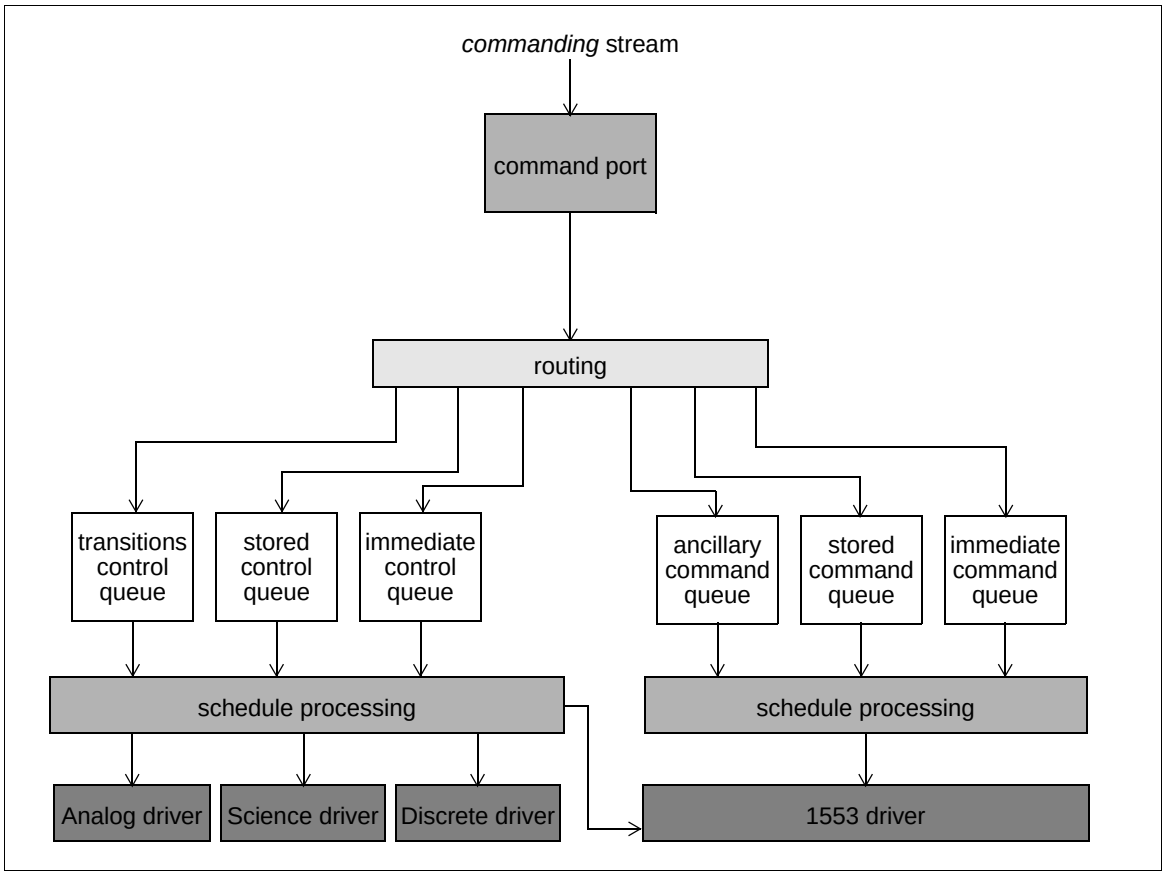

Figure 4 Command and Control flow through the VSC

1.4.1 Routing

The VSC receives direction through a specific type of telecommand called a *control request* (see Section 7.4). Requests to the VSC originate from two different sources:

- From the *Proxy*, through a network interface. Requests are transmitted from the proxy up to the VSC, through the *command* stream. Requests arriving on the command stream are meant to provide direction, either to the VSC, or indirectly the LAT.
- From the LAT. These requests arrive from the LAT through the 1553 interface.

Once requests arrive at the vsc, the router (based on request contents) directs the request to its appropriate queue.

1.4.2 Queuing

Queues are used to buffer requests dispatched from the router[1]. Nominally, the queue treats a request as a container holding a variable number of telecommands. Each telecommand represents a quanta of work requested of the VSC. The queue's principal responsibility is to decouple the rate at which the requests arrive with respect to the rate at which work (as represented by the telecommands) can be performed. The queue serves the additional function of storing requests to be executed at a future, *specific* time.

A queue contains two lists: A *pending* and a *free* list. Lists have FIFO discipline. An entry on either list is a reference to a *work* request. If a work request is on the pending list, it represents work to be performed. If a work request is on the free list, it is available as a template from which to form a pending work request. The work request contains a variable number of telecommands up to some predefined maximum. A work request also specifies the *time* at which the actions represented by the set of telecommands are to be executed. *Execution* time is specified using GLAST standard convention, where time is represented as the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The relationship between list and work request is illustrated in Figure 5:

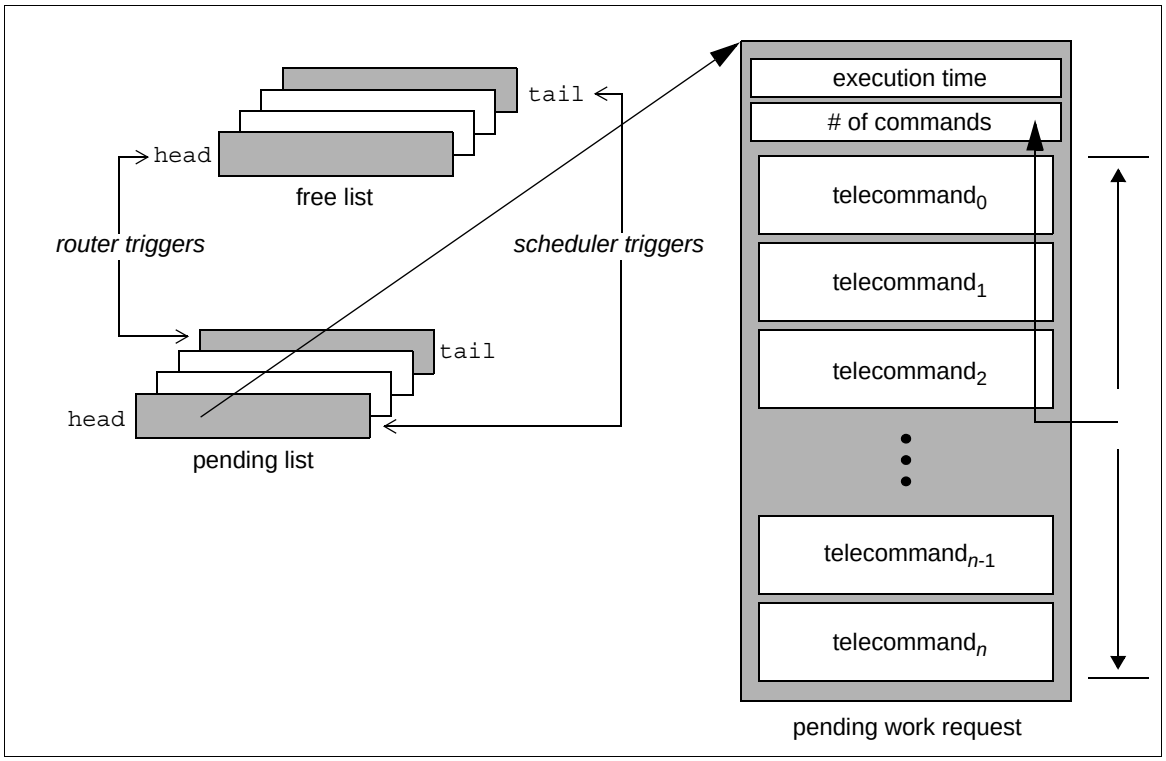

Figure 5 VSC queue, both immediate and stored.

The life cycle of a work request is basically as follows: when a request telecommand from the router arrives, a work request is removed from the head of the free list, the contents of the router's request copied to the work request, and the work request inserted at the tail of the

1. With the exception of LAT based requests, which originate through the 1553 interface.

pending list. At an appropriate time, the scheduler (as described in Section 1.4.3), removes a work request from the head of the pending list, performs the corresponding work and then returns the work request by inserting at the tail of the free-list. There are seven different activities which compete for the scheduler's resources and thus the VSC assigns to each activity its own queue. Each queue is differentiated by three parameters:

i. The maximum number of telecommands held by any one work request.

ii. Sort order. Entries may be ordered either in arrival time, or execution time. A queue which sorts by arrival time is called an *immediate* queue. A queue which sorts by execution time is called a *stored* queue.

iii. The maximum number of pending work requests. Each queue has default value, but this value can be user configured (see Section 3.8.1).

The parameterization of these seven queues is enumerated within Table 4:

Table 4 Queue scheduling order

name	type	Contains...	maximum[1]	size[2]
`Transistion`	*Immediate*	VSC scheduler state transitions	`1`	`16`
`ControlStored`	*Stored*	VSC control telecommands	`7`	`16`
`Control`	*Immediate*	VSC control telecommands	`7`	`16`
`Ancilliary`	*Stored*	LAT ancillary telecommands	`7`	`8192`
`CommandStored`	*Stored*	LAT telecommands	`8`	`16`
`Command`	*Immediate*	LAT telecommands	`5`	`16`

1. Expressed as the maximum number of telecommands per work request.
2. This is the default size, expressed as the maximum number of entries.

The sum of all work requests, over all queues represent the work that the VSC is asked to both schedule and execute.

1.4.3 Scheduling

The scheduler is woken up periodically, once per second. Its principal responsibility is to both determine and then sequence any necessary work within each one second *period*. The process requires the scheduler to break up each period into twenty-five, *40* milliseconds time *slots* as illustrated in Figure 6:

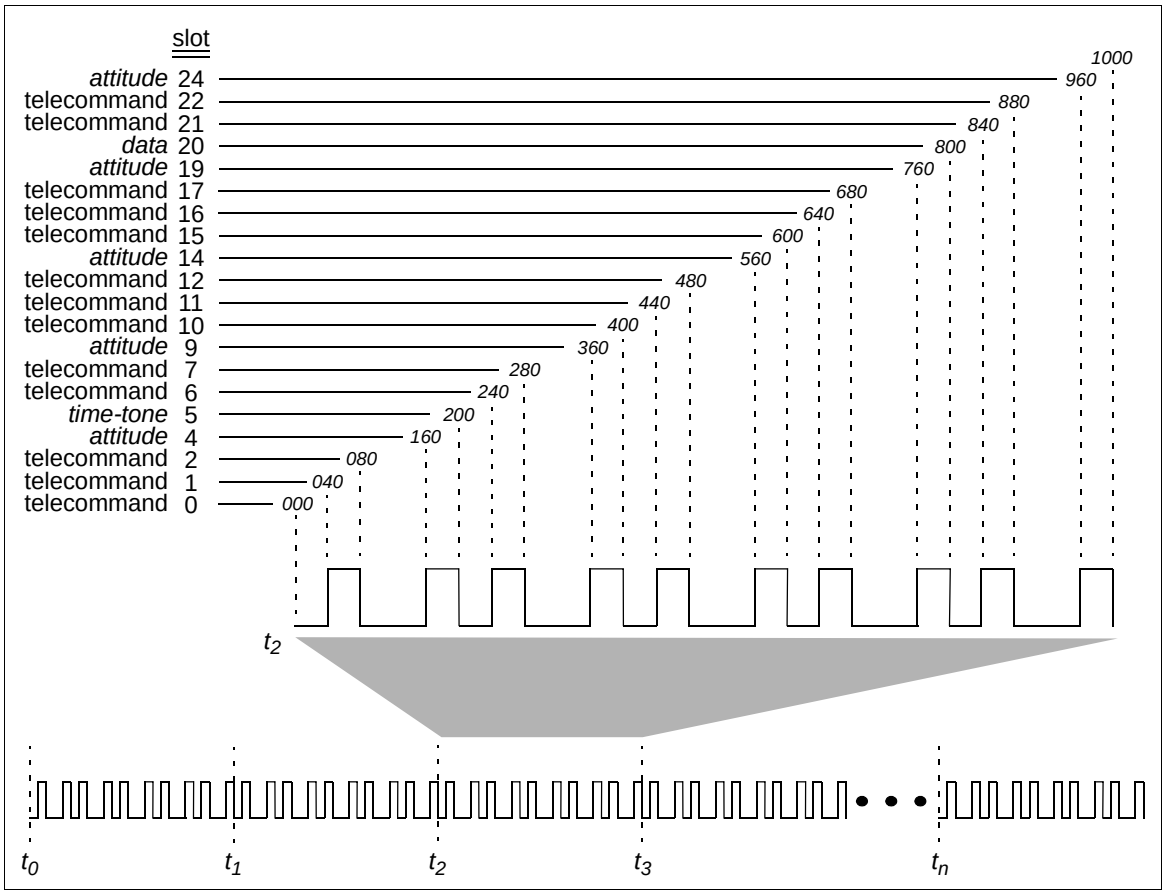

Figure 6 Scheduling within a period

This figure shows the succession of periods (labelled t_0, t_1, etc...) and the structure of any one period (for this example, period t_2). Within the period are the twenty-five slots, numbered such that increasing slot number corresponds to increasing time (relative to the period's start). Associated with each slot, is a potential telecommand to be sent by the VSC to the LAT (see Section 1.4.3.4). For example, slot *nine* (9) is offset *360* milliseconds from the beginning of the period, persists until *400* milliseconds, and is associated with the transmission of an ancillary *attitude* telecommand (see Section 4.10). Spanning these twenty-five slots, scheduling involves five distinct *phases*, sequenced as illustrated in Figure 7:

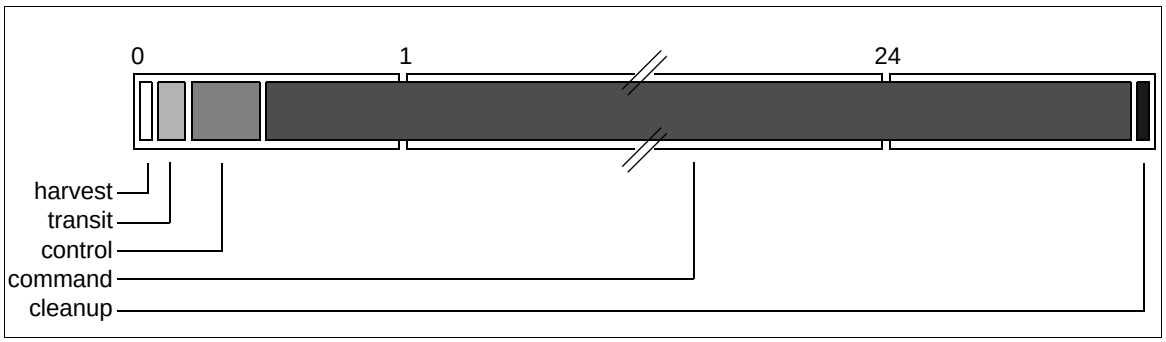

Figure 7 Phases within a period

The activity within each of these phases is described below.

1.4.3.1 Harvesting the work load

During this phase each of the seven queues are probed for potential work. If a queue is not empty, the *due* field for the work request at the queue's head is examined. The due field contains either *zero* (0) or an absolute time corresponding to the period in which the work contained within the entry should be performed. A value of zero specifies the entry contains *immediate* work. Immediate work, independent of period, is scheduled as soon as it appears[1]. A non-zero value specifies the entry contains *stored* work. If this value is non-zero, it is compared to the period's time. If equal, the work contained within the entry should be scheduled. In short, from the scheduler's perspective, harvesting consists of requesting from the queue its list of work for that period. The work list is a series of telecommands, each telecommand representing one quanta of work. Therefore, at the end of harvesting, the scheduler has seven different work lists, one for each queue. If the queue does not have any work for the specified period its work list will be empty. This phase is the first activity within slot *zero* (0).

1.4.3.2 Scheduling state transitions

During this phase the scheduler attempts to process the telecommands on the work list associated with the *transition* queue (see Table 4). The scheduler's behaviour is modelled as a Finite State Machine (FSM). The states and transitions of this machine are discussed in Section 1.4.4. The events which trigger state transitions correspond to the telecommands on this work-list. Therefore, during this phase the scheduler transits to a potentially new state consistent with its current state and the presence of any transition events. Once this phase completes, if the scheduler is left in either the *paused* or *stopped* state the remainder of the phases discussed below are *not* executed. This phase occurs within slot *zero* (0).

1.4.3.3 Scheduling control work

During this phase the scheduler attempts to process the telecommands on the work lists associated with the LAT requests, immediate, and stored *control* queues (see Table 4). Control telecommands provide direction to the VSC. For example, a request to download diagnostic telemetry (see xxx for a list of these telecommands). In order to bound the time used by this phase, the maximum number of control telecommands which can be executed during any one period is arbitrarily limited to *eight* (8), however, the sum of the telecommands on the three lists could be as many as sixteen 16). The schedular priorities telecommand execution by processing the commands on the stored work list first, followed by the commands on the immediate work list and last by the commands on the LAT work list. The number of immediate commands executed is the minimum of the difference between the maximum (8)

1. Rounded to a period boundary.

and the number of stored commands processed and the number of commands on the immediate work list. any commands on the immediate which cannot be processed will be deferred to the subsequent period. Note that at most *seven* (7) telecommands can be on a stored work list. This ensures that, independent of the number of stored telecommands, at least one immediate telecommand is executed per period. This phase occurs within slot *zero* (0).

1.4.3.4 Scheduling command work

During this phase the scheduler attempts to queue to the LAT, through the VSC's 1553 interface, the telecommands on the work lists associated with the ancillary, immediate, and stored *command* queues (see Table 4). These telecommands provide direction to the LAT. For example, the telecommand used to take the SIU out of primary boot (see [8] for a list of these telecommands). The spacecraft (and consequently VSC) limits transmission to, at most, one command per slot. Five of these slots are not accessible to the user[1], therefore, the maximum number of proxy originated commands which can be sent to the LAT during any one period is limited to *twenty*. Seven of these slots are pre-allocated to the ancillary sequence (the so-called "magic seven") and each particular ancillary command is mapped to a specific slot. For example, the timetone command is always sent on slot *five*. (see Figure 6 for the complete assignments). This leaves *thirteen* slots allocated to arbitrary commands, to be scattered within arbitrary slots. The scheduler fills any given slot depending on whether the slot would be occupied with either an ancillary or generic command:

— If the slot specifies an ancillary command, the telecommand at the head of the work list associated with the ancillary queue is removed and transmitted. If, when removing a command from this list the list was empty, a default command is looked for. If the default is present, it is transmitted. If a default command is not found, the slot is left empty.

— If the slot specifies an generic command, the telecommand at the head of the work list associated with the *stored* command queue is removed and transmitted. If, when removing a command from this list the list is empty, the telecommand at the head of the work list associated with the *immediate* command queue is removed and transmitted. If, when removing a command from this list the list is empty, the slot is left unfilled.

This phase occurs on slots *zero* (0), through slot *twenty-four* (24).

1.4.3.5 Cleanup

This phase occurs after all the work scheduled for this period is performed. In this phase the work requested of the period is compared with the work performed within the period. When the two are equal, the resources required to perform the work are recovered. In detail this involves asking each of the six queues to examine the work-list for the work request at the head of its pending list. If the work-list is empty, the work request is removed from the queue's pending list and returned to its free-list. This phase occurs on the *last* slot (24).

1. They are reserved to the 1553 hardware/driver to solicit telemetry from the LAT.

1.4.4 Scheduler State Model

The scheduler has abstract behaviour. For example, it can be started or stopped. This behaviour is modelled as a Finite State Machine (FSM) with five *states*. These states are:

1. *Stopped*. In this state:
   - all queues are nominally *empty*
   - the VSC *accepts* all telecommands
   - time does *not* increment
   - scheduling is *disabled*
2. *Paused*. In this state:
   - the state of all queues is *irrelevant*
   - the VSC *accepts* all telecommands
   - time does *not* increment
   - scheduling is *disabled*
3. *Running*. In this state:
   - the state of all queues is *irrelevant*
   - the VSC *accepts* all telecommands
   - time increments
   - scheduling is *enabled*
4. *Flushing queues*. In this state:
   - the state of all queues is *indeterminate*
   - the VSC *rejects* all telecommands
   - time does *not* increment
   - scheduling is *disabled*
5. *Setting time*. In this state:
   - the state of all queues is *irrelevant*
   - the VSC *rejects* all telecommands
   - time does *not* increment
   - scheduling is *disabled*

Transitions between states are triggered by the execution of a scheduler request. Commands of this type are scheduled for execution within the phase discussed in Section 1.4.3.2. This command has two functions:

State transition: Requests a state transition. The function code enumerates the new state requested of the scheduler. The telecommand parameter is ignored. The set of potential state transitions are:

— *Start*, put the VSC into a state of running.

— *Pause*, put the VSC into a paused state.

— *Stop*, put the VSC into a stopped state.

Set Time: Request a modification of the VSC's time-base (see Section 1.3). The telecommand parameter specifies the new value of the time-base.

Like all telecommands sent on the command stream, the user queues these telecommands through the proxy interface. In this particular case, the interface is contained within the `Proxy` class discussed in Section 7.2. Figure 8 illustrates the combination of states and permitted transitions:

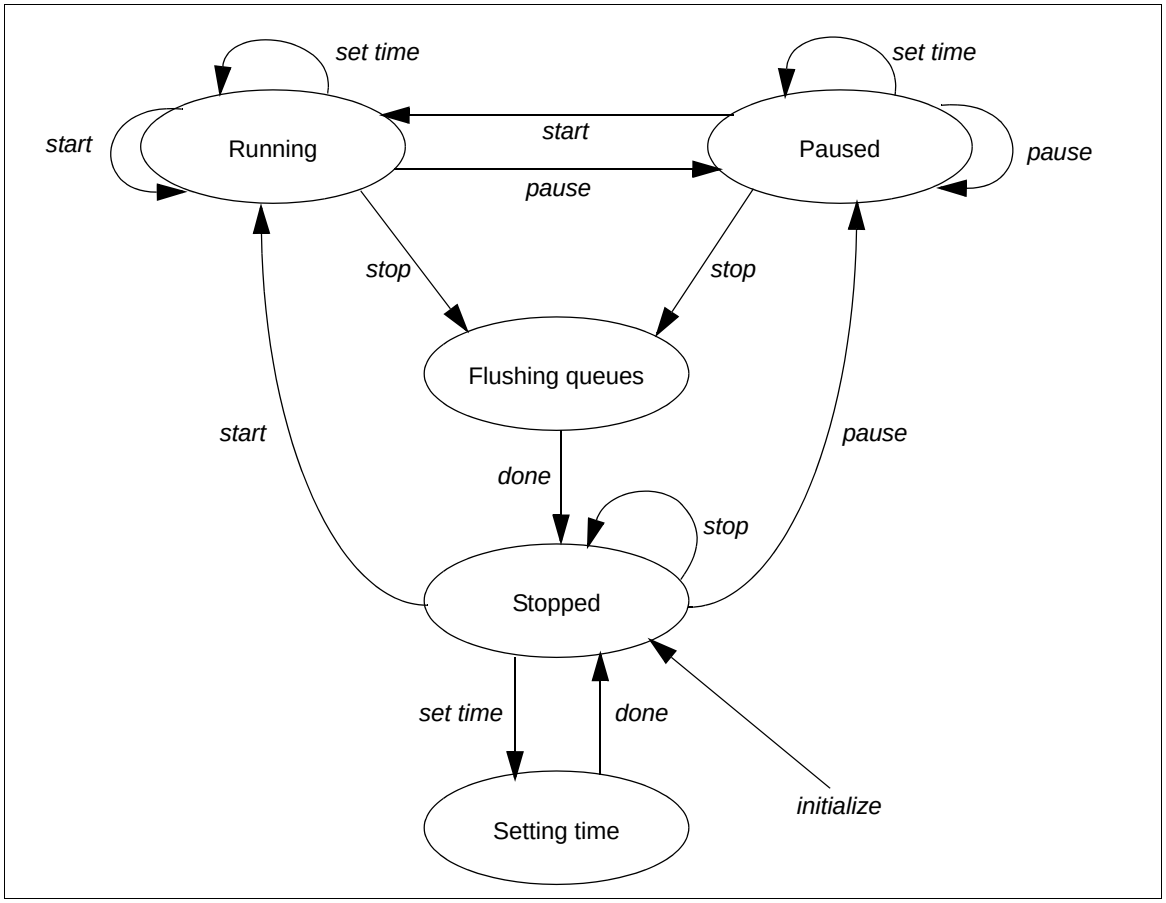

Figure 8 VSC Finite State Machine Diagram

To summarize the main features of this diagram:

— The scheduler comes up in a *Stopped* state

— Two of these states (*Flushing queues* and *Setting time*) are transitory states

— One can only modify the time-base in the *Stopped* state

— The principle difference between the *Paused* and *Stopped* states is that a transition to the stopped state will flush the queues.

Chapter 2
Hardware

Physically, the VSC consists of up to eight different types of modules. Seven of the modules are VME modules and the eighth is a PMC card. These modules are:

i. A commercial Single-Board Computer (SBC). The SBC is a VME module, model Motorola MVE2304 (see xxx). The 2304 serves two functions: First, as the platform under which the software of the VSC executes (see chapter 3), and, second as the carrier for the 1553 PMC cards (see below).

ii. A commercial *1553* interface. The 1553 interface is a model PMC-1553B from *Alphi Technologies* (see xxx). This interface is on a PMC form factor and resides in the SBC described above. A fully populated VSC contains two of these interfaces. One serving as the *primary* 1553 interface and the second as the *redundant* 1553 interface.

iii. A commercial GPS receiver. The GPS receiver is a VME module, model TTM637VME from *Symmetricom Industries* (see xxx). This module serves as the time-base for the VSC and consequently for the LAT when it is attached to the VSC.

iv. A *Science* interface. This interface is a in-house produced, VME module called the VSC-SCI. A fully populated VSC contains two of these modules. One serving as the Primary science interface and the second as the Redundant science interface.

v. A *Discrete* interface. This interface is a in-house produced, VME module called the VSC-DSC. A single module implements both the *primary* and *redundant* discrete interfaces.

vi. One or more commercial digitizer boards. The digitizer board is an industry standard I/O pack, model VMESC5, from *Systran Industries* (see xxx). These boards digitize the 96 quantities sources by the LAT, but monitored by the VSC.

vii. A *cable* interface to the LAT. This interface is a in-house produced, VME module called the VSCIO-850. This board is one of two cable interface modules (see below) between the VSC and LAT. This particular module services all singles which either come from or go to the LAT shield. This module has two functions: It adapts the cabling from the shield to the flat ribbon cables required by the appropriate interface modules.

2.1 Electrical conventions

2.2 The SBC and 1553 interface

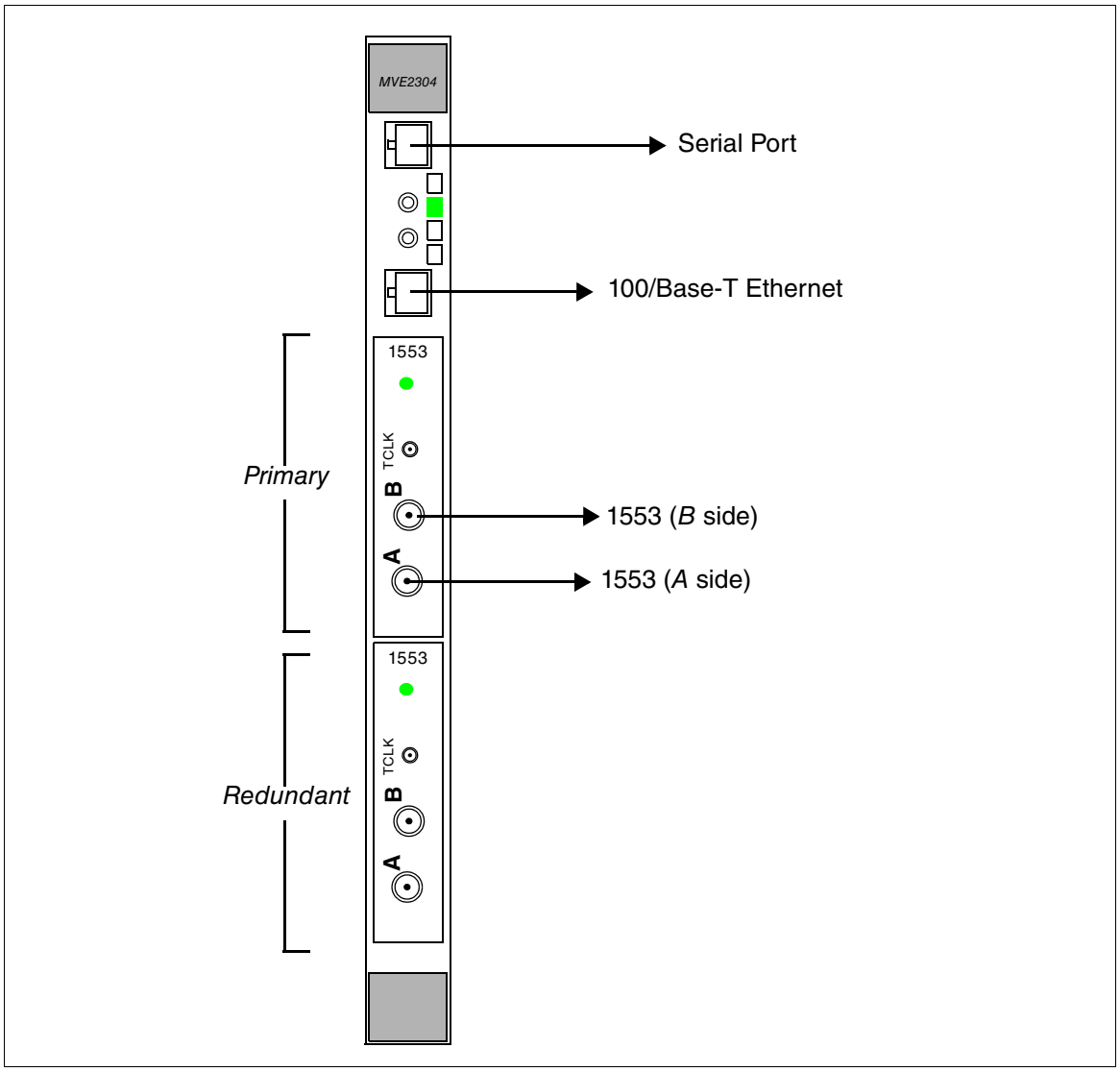

Figure 9 SBC and 1553 interface

2.3 The GPS receiver

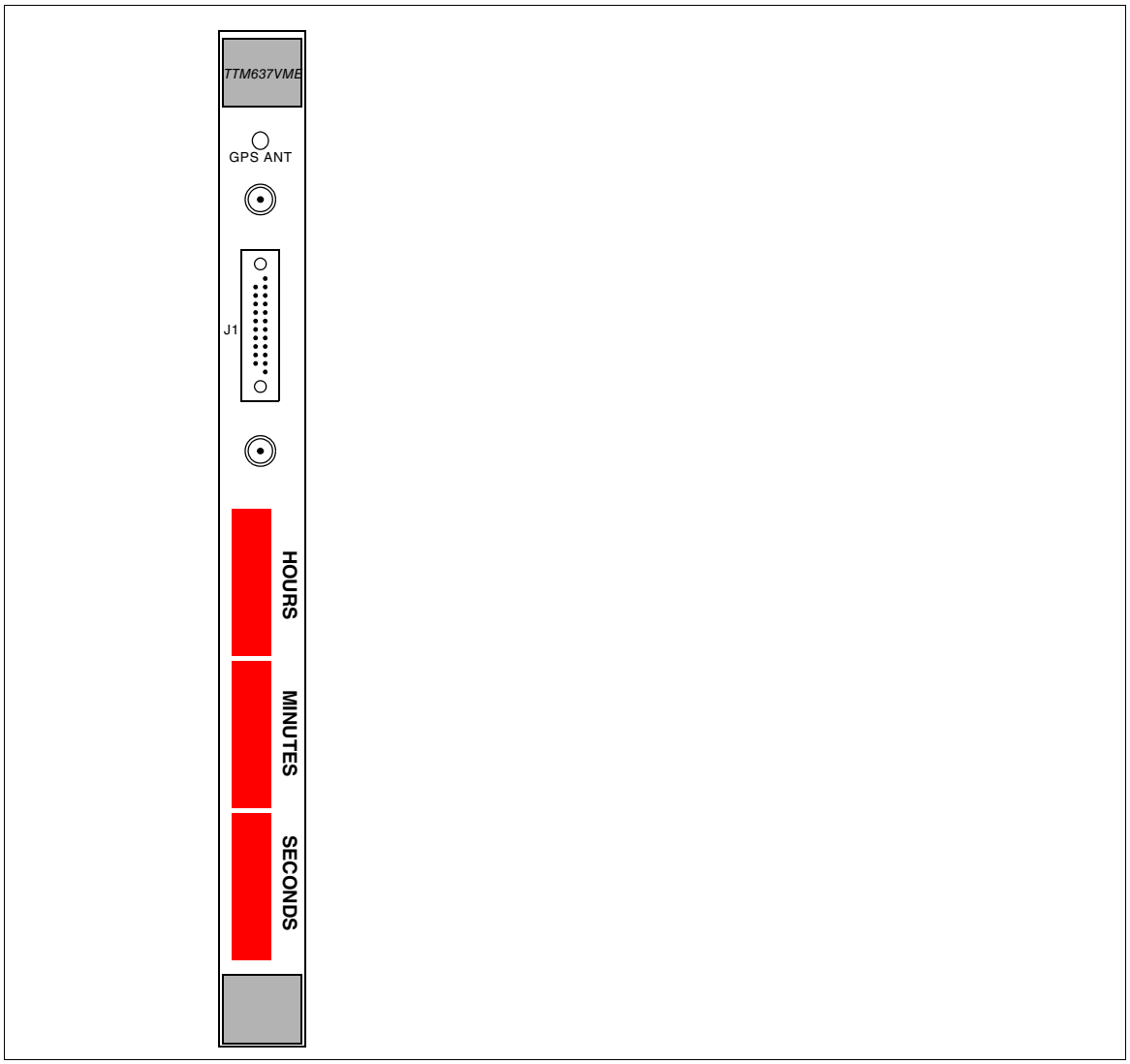

Figure 10 GPS receiver

2.4 The Science interface

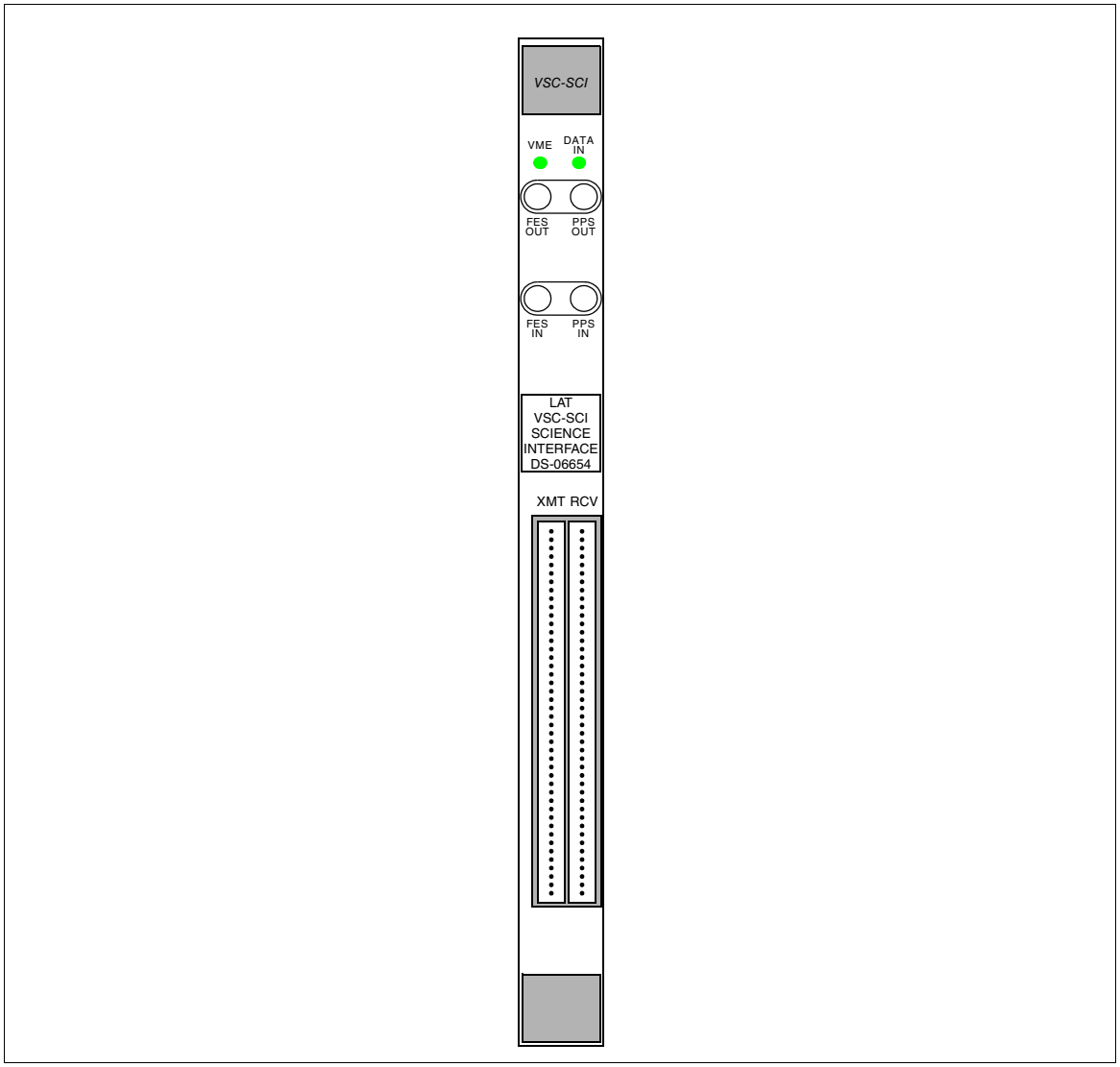

Figure 11 Science interface

2.5 The Discrete interface

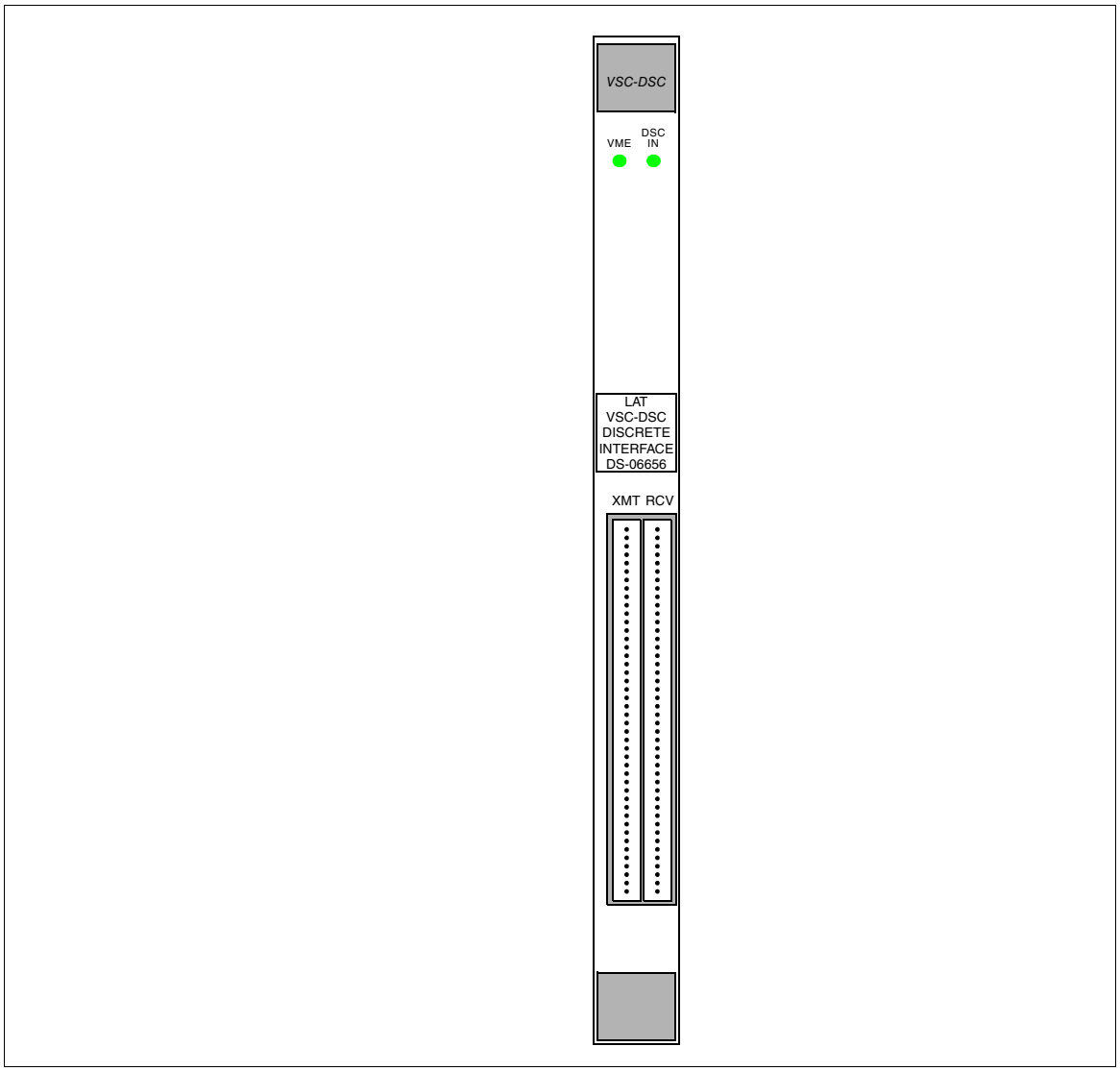

Figure 12 Vsc components

2.6 Digitizer

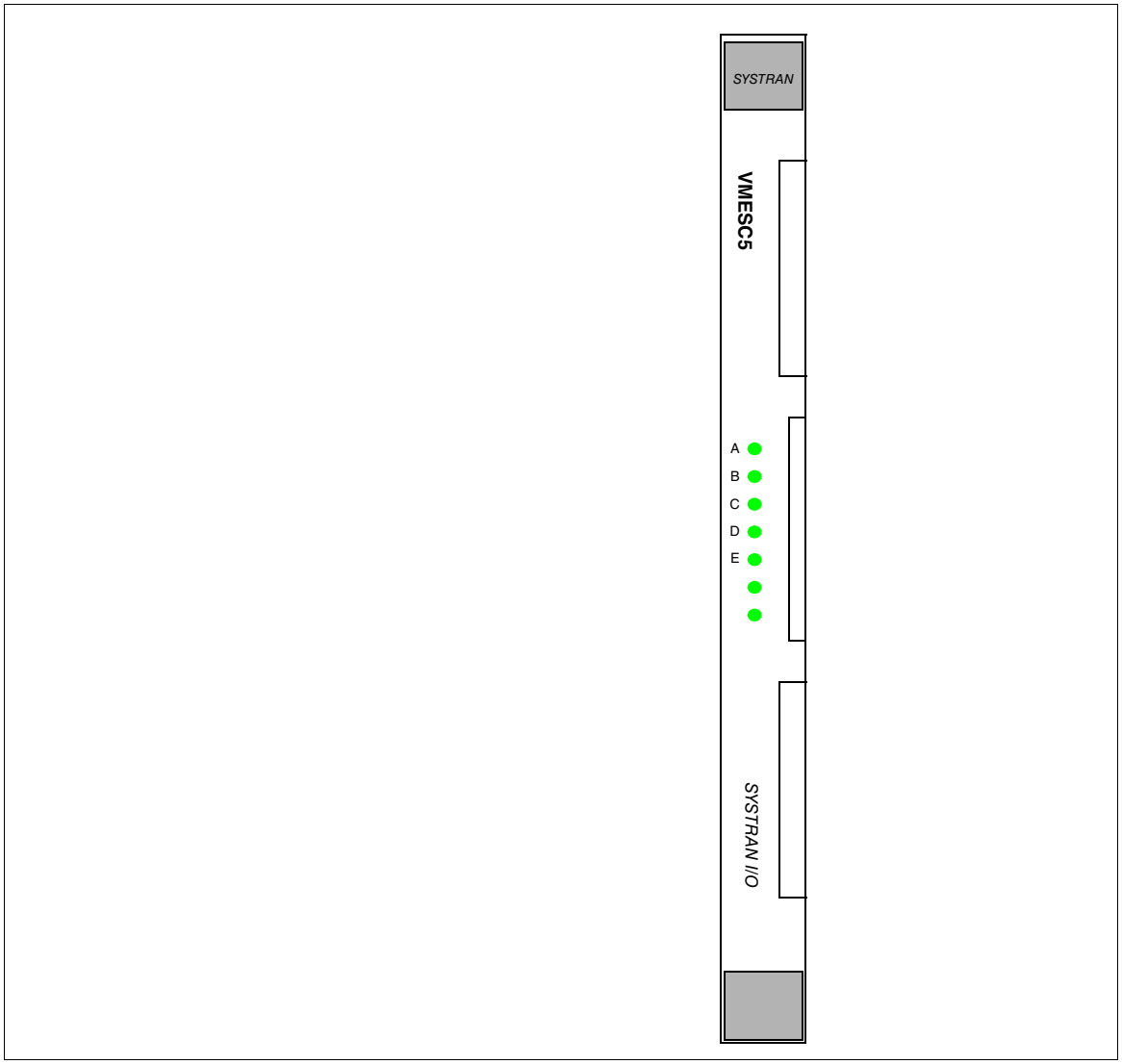

Figure 13 Digitizer

2.7 LAT Cable interface

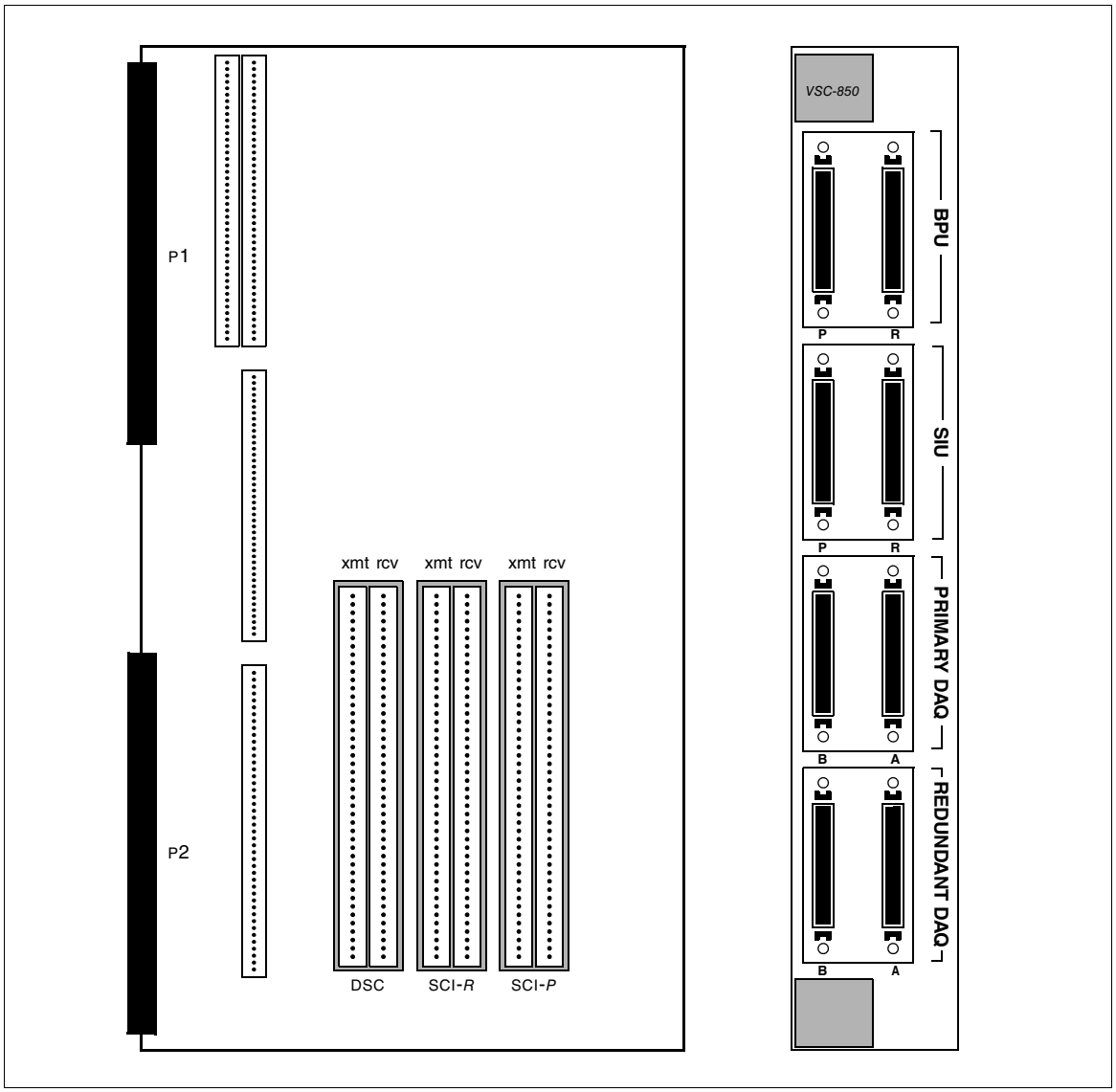

Figure 14 VSC-850 module

2.8 LAT Monitoring interface

2.9 VSC Configurations

2.9.1 The Testbed

As shown in Figure 15, the *testbed* configuration consists of the following modules:

- One (1) 21-slot 6U VME crate
- One (1) MVE2304 Single-Board-Computer
- Two (2) PMC-1553B 1553 interface modules
- One (1) TTM637VME GPS receiver
- Two (2) VSC-SCI science modules
- One (1) VSC-DSC discrete interface module
- One (1) *Systran* I/O substrate with four (4) industrial packets
- One VSC-850 LAT interface module

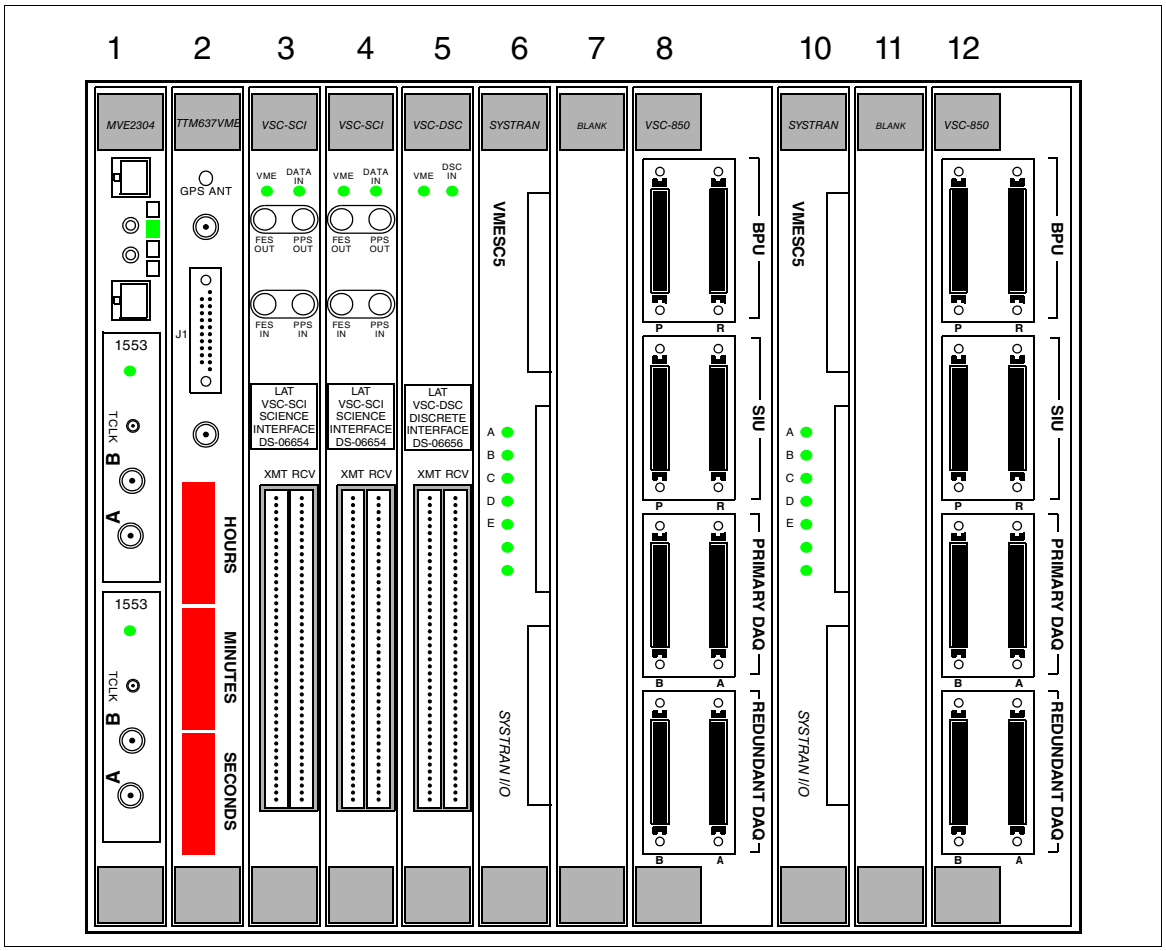

Figure 15 Corner teststand configuration

Figure 16 illustrates how these modules are cabled up. This configuration requires the following cables:

— Six (6) LAT-DS-xxxxx

— Six (4) LAT-DS-xxxxx

— Six (5) LAT-DS-xxxxx

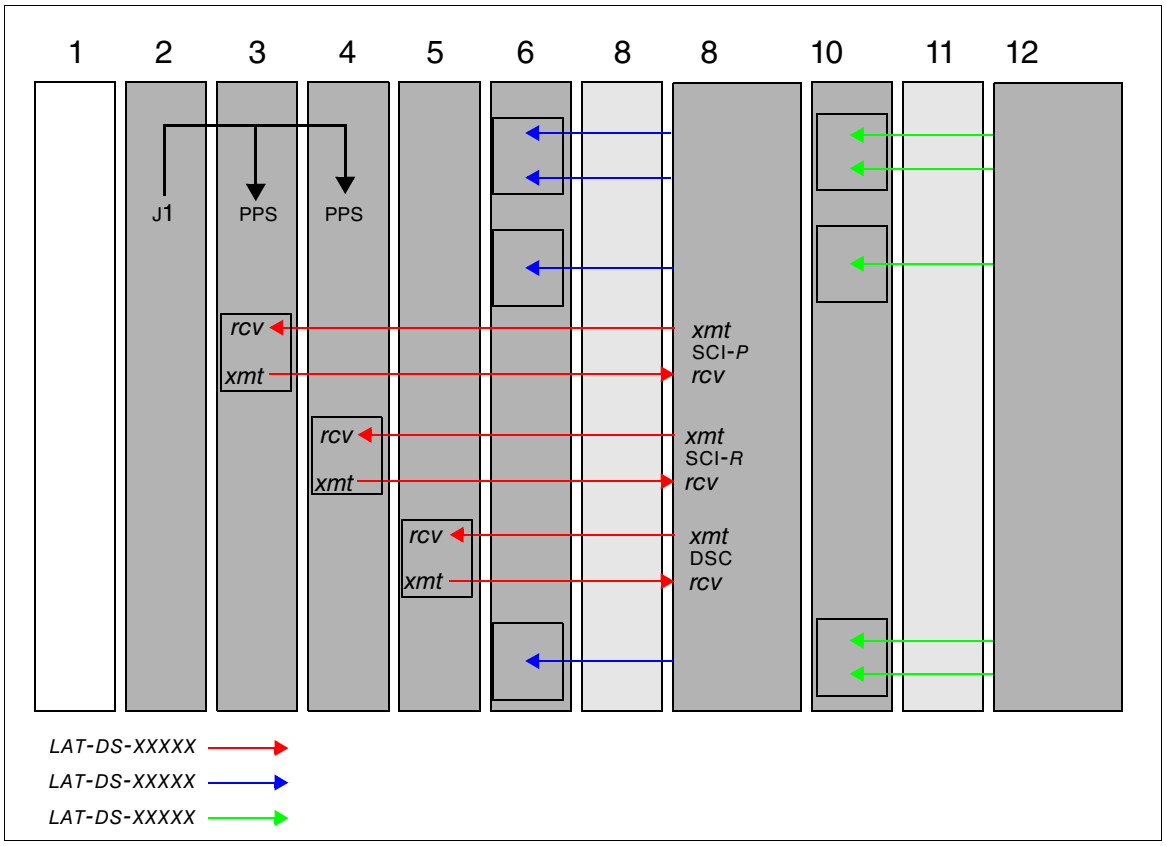

Figure 16 Internal Connectivity (LAT VSC)

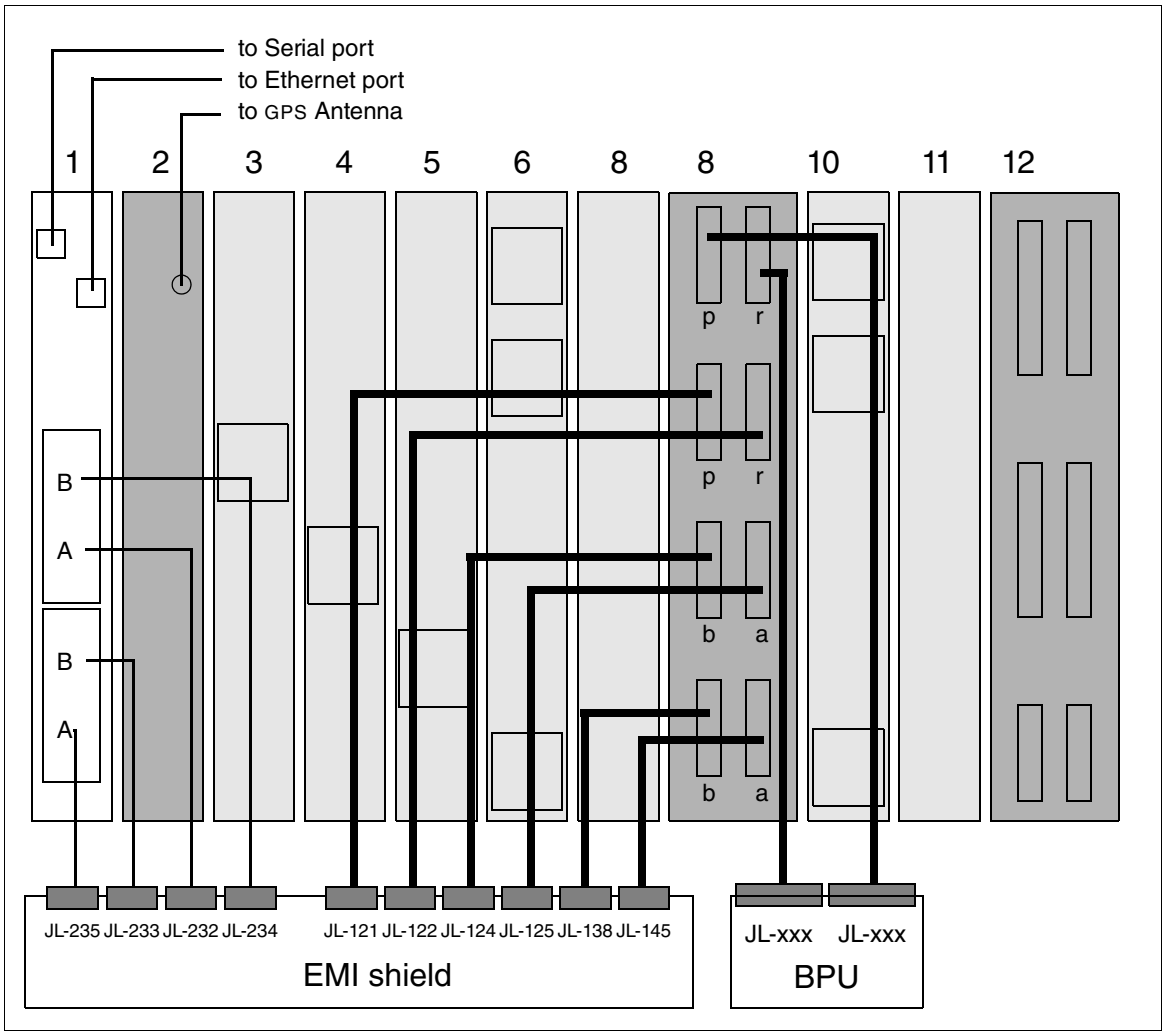

Figure 17 External Connectivity (LAT VSC)

2.10 VSC Inter-Connectivity

Chapter 3
Using the proxy interface

Before starting any application assembled from the proxy interface, the application user should be satisfied on two counts:

i. There is agreement between the VSC and proxy interface on the VSC's host name and the port numbers used for communication. This subject is addressed in Section 3.8.

ii. The VSC is successfully booted (see Section 3.8).

Once the VSC is successfully booted, the user's application is ready to control the VSC through the proxy interface. To do so, first, the application must establish a connection to the VSC. This is very straightforward and requires only an instantiation of a `Proxy` object (see Section 7.2). For example:

```
Proxy vsc;
```

If the `Proxy` class is successfully instantiated, the application is successfully connected to the VSC. Once connected, the application controls the VSC by instantiating one or more requests or telecommands, and then invoking the `execute` member of the `Proxy` class (see Section 7.2), passing the request as an argument. For example:

```
vsc.execute(Grb);
```

constructs an object which instructs the VSC to issue to the LAT a GRB (Gamma-Ray Burst) alert (`Grb`, see Section 7.13) and then transmits this request to the VSC for later execution.

Note the VSC comes up in the *stopped* state (see Section 1.4) and needs starting before any commands queued to the VSC are actually executed (including the previous example). However, before starting the VSC, a user typically needs to perform some application specific initilization. While the form of this initilization is peculiar to each application, most likely it would include the following steps:

— for each type of telemetry define its router

— define an appropriate set of telemetry handlers
— register an appropriate set of telemetry routers
— queue an initial set of LAT telecommands
— establish the VSC's time
— start the scheduler

The following code fragments provide one, rather pedagogical, example of how the VSC could be initialized. First, the necessary telemetry routers must be defined, instantiated and registered with the proxy interface. For example, assume (albeit, unrealistically) the router defined in Section 3.5 could be used to catch and process all incoming VSC and LAT telemetry. Then to instantiate and register this router...

```
MyRouter* router = new MyRouter;

vsc.latDiagnostic(*router);
vsc.vscDiagnostic(*router);
vsc.latTelemetry(*router);
vsc.vscTelemetry(*router);
vsc.latScience(*router);
```

Once registered, the application must be prepared to catch and process telemetry. Note that while VSC telemetry is inhibited with the scheduler stopped, there is no such guarantee for LAT telemetry. While highly unlikely that telemetry could flow out of the LAT and through the VSC with the scheduler stopped, there is nothing in the VSC to actually prohibit such an action. So if commanding (or any other initilization) should be live *before* telemetry is received, the application could defer registration until the last step of initilization.

Next, any LAT telecommands that the application wishes present *before* scheduling is started are queued. For example, assume, before any user interaction with the LAT should be allowed, the application requires assurance that the LAT 's housekeeping system is in a well known state. The LAT initializes the housekeeping system in response to a `SYSRET` telecommand. A class representation of this telecommand is defined in Section 3.6. Therefore, to initialize housekeeping one simply instantiates this class, invokes the `execute` member of the proxy interface, passing as an argument the instantiated telecommand:

```
vsc.execute(SysReset);
```

The `execute` member transmits the specified telecommand to the VSC, where it is routed to its appropriate queue for execution at the time until the scheduler is started. Once, the VSC is started (see below), the telecommand is then set to the LAT, through its 1553 interface. Note the `execute` function is *synchronous*, the function does not return to the caller until the command has either been safely transmitted to the VSC, or an exception has been thrown.

The next step in the initialization requires the setting the absolute time for the VSC (and consequently the LAT):

```
#define T0 ((unsigned)0xDEADBEEF)

vsc.execute(SetTime(T0));
```

Need some more explanation here.

Next, the scheduler (and the VSC's clock) is started:

```
vsc.execute(Scheduler(Start));
```

Finally, the LAT itself is reset:

```
vsc.execute(Reset);
```

Note, that because the VSC allows more then one command to be executed each period, the three commands of the examples above could have been queued simultaneously:

```
vsc.execute(Scheduler(Start), SetTime(T0), Reset);
```

3.1 Distribution of telemetry

The preceding example connects all five telemetry streams with one instantiation of the proxy interface on one host. Schematically this scenario is represented by Figure 18.

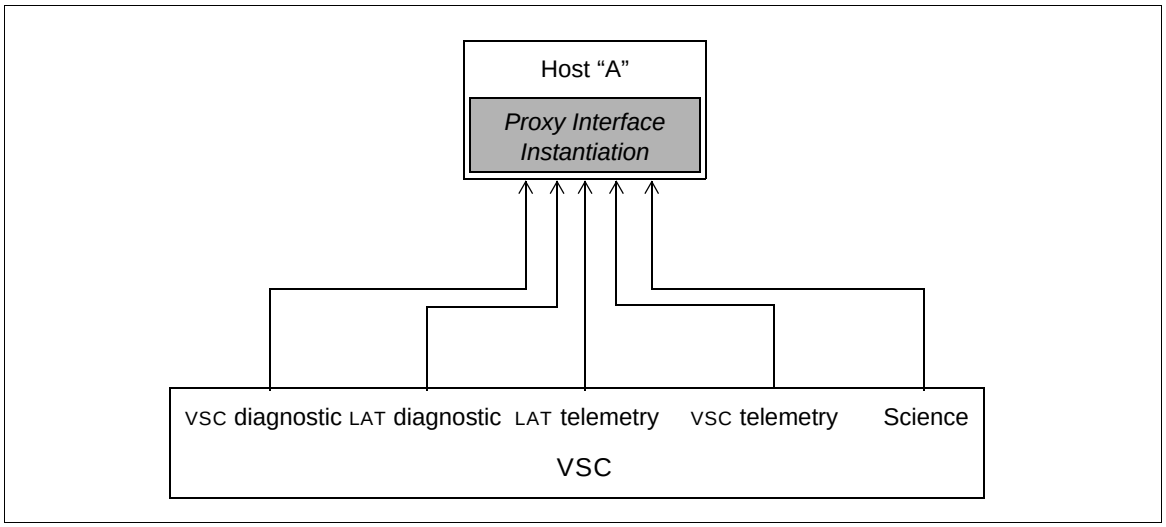

Figure 18 One proxy interface processing all telemetry streams

However, there is no necessity to register all streams with any one instantiation of the proxy interface. For example, diagnostic telemetry is considered “real-time” and its prompt delivery may be useful to the robust operation of the VSC. In such a case, dividing the diagnostic telemetry from the housekeeping and science telemetry by directing the streams to two different proxies could be attractive. One proxy would be responsible for the control and monitoring of the VSC and the other proxy responsible for processing any data delivered by the VSC. An even greater isolation between these functions could be achieved by instantiating the proxies on separate hosts. Such as scenario is represented in Figure 19:

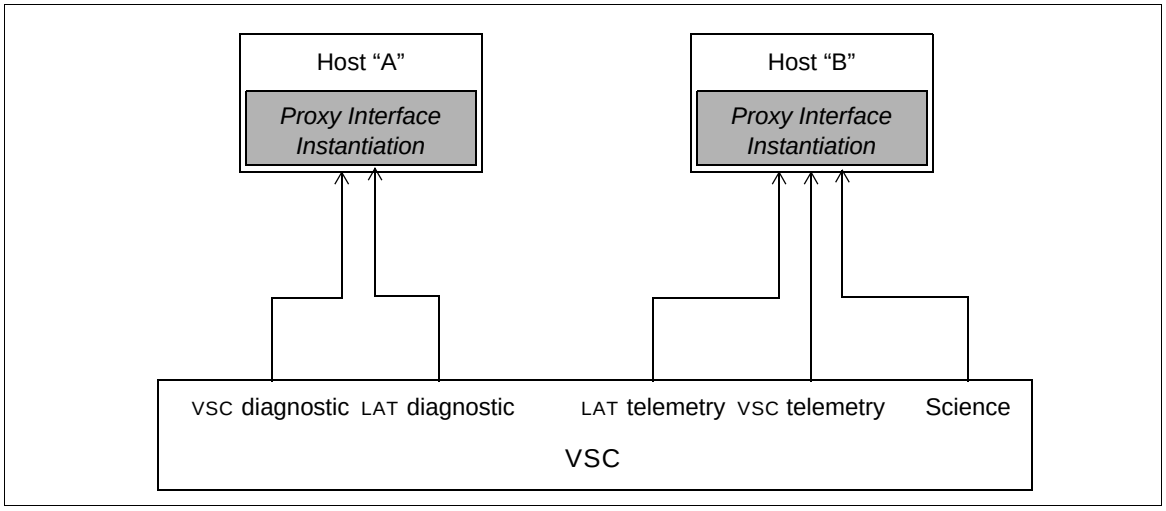

Figure 19 Multiple proxy interfaces

In this scenario, the user on host “A” would connect as follows:

```
MyRouter* router = new MyRouter;

Proxy controlAndMonitor;

controlAndMonitor.latDiagnostic(*router);
controlAndMonitor.vscDiagnostic(*router);
```

While a user on host “B” would connect as follows:

```
MyRouter* router = new MyRouter;

Proxy processor;

processor.latTelemetry(*router);
processor.vscTelemetry(*router);
processor.latScience(*router);
```

Note, that independent of number of proxies instantiated, or which proxies are connected to which telemetry streams, each proxy retains a connection to the command stream and consequently the ability to command the VSC. Therefore, an application must be carefully crafted when apportioning control responsibilities in order to insure the two users do not conflict in the management of the VSC.

3.2 Managing the SC/GBM interface

The GBM instrument’s primary responsibility with respect to the LAT is notification of candidate GRBs (Gamma-Ray-Burst). As the GBM does not communicate directly with the LAT, these alerts are relayed through the spacecraft. When notified by the GBM, the spacecraft sends an agreed upon signal to the LAT. This interface allows an application to trigger the VSC to emit such a telecommand. For example:

```
vsc.execute(Grb);
```

3.3 Managing the SC/LAT interfaces

3.3.1 SIU cross-strapping

The LAT has two SIUs (Spacecraft Interface Unit). One SIU is designated as the *Primary* SIU and the other as the *Redundant* SIU. In orbit, the redundant SIU is called out as a cold spare. As the name implies, SIUs interact with the spacecraft. In turn, the spacecraft mitigates against single-point failure by having *two* SIU interfaces which the VSC emulates. One interface is called out as the *Primary* LAT interface and the other as the *Redundant* LAT interface. In order to support full redundancy each of the four units has both an "A" and "B" port. This results in the cross-strapping of both interface and SIU as illustrated by Figure 20:

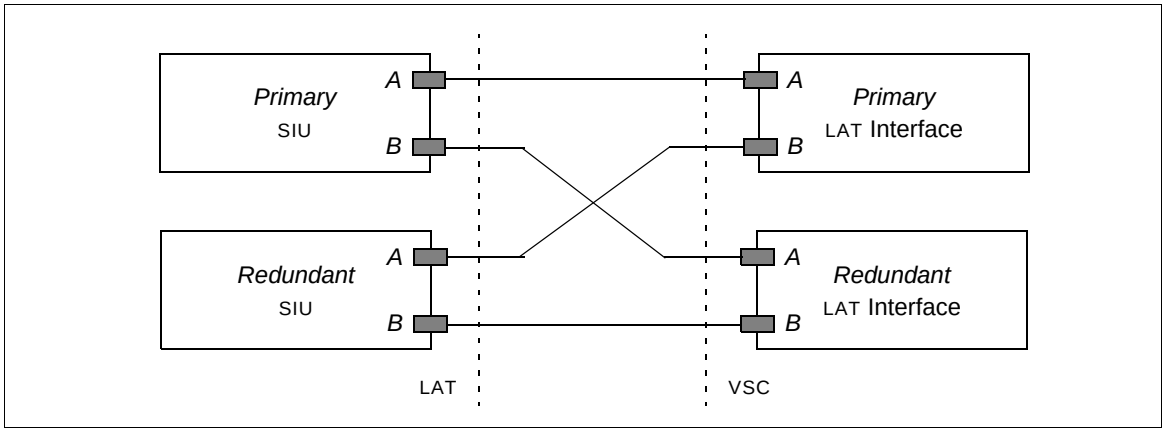

Figure 20 SIU cross-strapping

The figure illustrates that any one time the spacecraft uses *one* of its interfaces to communicate with *one* of the LAT's SIUs. Note that while the figure shows any one communication path as a single "wire", in actual fact, the wire represents communication between the three different types of physical interfaces between SIU and VSC. These include:

i. the 1553 interface
ii. the Discrete interface
iii. the Reset interface[1]

Therefore, whenever the cross-strapping is changed for one, it's changed in unison for all three of these physical interfaces. Each one of the four options corresponds to a particular path between the units as enumerated in Table 5:

1. Which is, from the viewpoint of the spacecraft, a part of the discrete interface.

Table 5 Cross-strapping options for the SIU interface

Path	Use Unit...			
	VSC		LAT	
	Primary	Redundant	Primary	Redundant
AA	yes	no	yes	no
AB	yes	no	no	yes
BA	no	yes	yes	no
BB	no	yes	no	yes

*When the VSC is initially booted, the VSC defaults to using the AA path, which corresponds to using the **Primary** LAT interface communicating with the **Primary** SIU.*

The `SiuInterface` request class (see Section 7.6) is used to modify the current spacecraft/LAT cross-strapping. The class's constructor takes as an argument an enumeration (see Section 7.5) which specifies which one of the four cross-strapping options should be selected. For example, to communicate with the redundant SIU through the VSC's redundant interface:

```
vsc.execute(SiuInterface(BB));
```

Or to restore the cross-strapping to its default setting...

```
vsc.execute(SiuInterface(AA));
```

3.3.2 SIU Reset

The `Reset` request class (see Section 7.9) is used to issue a LAT *Reset*, as this fragment illustrates:

```
vsc.execute(Reset);
```

This reset signal is communicated to the LAT through whatever cross-strapping is currently established (see 3.3.1).

3.3.3 SIU discretes

The ICD specifies three[1] individual lines between the spacecraft and LAT which are *driven* by the spacecraft. The spacecraft can either *assert* or *deassert* these lines.

When the VSC is initially booted the VSC interface ***deasserts*** *the discrete lines.*

The `ToggleLines` request class (see Section 7.8) is used to toggle the state of the three discrete lines. In a single request the interface may change the value of one, two, or three of these lines. The first argument specifies the discrete lines to toggle and the second argument determines each line's new value. Each bit offset of each argument corresponds to a particular line. For example, bit-offset *zero* (0) corresponds to line *zero*. Only if a bit-offset in the first argument is *set*, will the corresponding line's value be changed. The new value of the line is determined by the corresponding bit-offset in the second argument. The `ToggleLines` class provides an enumeration for each of these offsets. For example, to *deassert* line *zero* (0) and to *assert* line *two* (2), while leaving line *one* (1) unchanged:

```
vsc.execute(ToggleLines(Line0 | Line2, ));
```

Which one of the two sets of discrete lines (primary or redundant) actually toggled is determined by whatever cross-strapping is currently established (see 3.3.1).

3.3.4 GASU (DAQ board) cross-strapping

The LAT has one GASU which contains two DAQ boards. One DAQ board is designated as the *Primary* DAQ board and the other as the *Redundant* DAQ board. Only one of the two DAQ boards is active at any one time. In turn, the spacecraft mitigates against single-point failure by having *two* DAQ board interfaces which the VSC emulates. One interface is called out as the *Primary* LAT interface and the other as the *Redundant* LAT interface. In order to support full redundancy each of the four units has both an "A" and "B" port. This results in the cross-strapping of both interface and DAQ boards as illustrated by Figure 21:

1. Actually *four*, but the fourth is used as the LAT reset and is called out separately in Section 3.3.2.

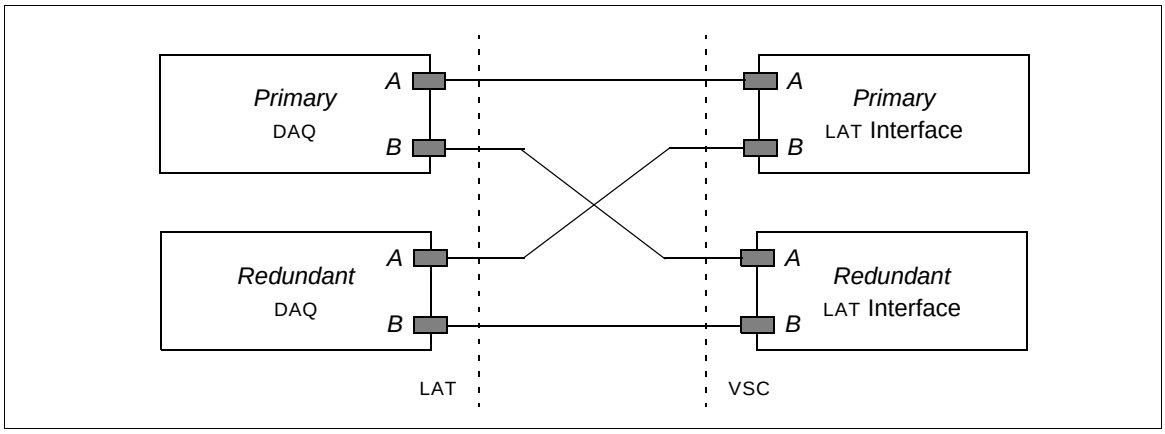

Figure 21 DAQ board cross-strapping

The figure illustrates that any one time the spacecraft uses *one* of its interfaces to communicate with *one* of the LAT's DAQ boards. Note that while the figure shows any one communication path as a single "wire", in actual fact, the wire represents communication between the three different types of physical interfaces between SIU and VSC. These include:

i. the science interface

ii. the 1-PPS interface

iii. the analog temperature and voltage measurements associated with the DAQ board

Therefore, whenever the cross-strapping changes, it's changed in unison for all three of these physical interfaces. Each one of the four options corresponds to a particular path between the units as is enumerated in Table 6:

Table 6 Cross-strapping options for the DAQ interface

	Use Unit...			
	VSC		LAT	
Path	Primary	Redundant	Primary	Redundant
AA	yes	no	yes	no
AB	yes	no	no	yes
BA	no	yes	yes	no
BB	no	yes	no	yes

*When the VSC is initially booted, the VSC defaults to using the AA path, which corresponds to using the **Primary** LAT interface communicating with the **Primary** DAQ board.*

The `DaqInterface` request class (see Section 7.7) is used to modify the current cross-strapping. The class's constructor takes as an argument an enumeration (see Section 7.5)

which specifies which one of the four cross-strapping options should be selected. For example, to communicate with the primary DAQ board through the VSC's redundant interface:

```
vsc.execute(DaqInterface(AB));
```

Or to restore the cross-strapping to its initial value...

```
vsc.execute(DaqInterface);
```

3.3.5 Enabling the Science Interface

The science interface on the spacecraft contains a data enable. This enable is the so-called "DEVICE_READY_LINE". In order to enable or disable the SSR interface one instantiates the Ssr class (see Section 7.11).

When the VSC is initially booted, the science interface is ***disabled****.*

Until the science interface is enabled any science data sent to the VSC by the LAT, remains queued on the LAT. Once the interface is enabled (the DEVICE_READY_LINE is asserted), this queued data will be received on the VSC and forwarded to the proxy interface as science telemetry (see [8]). For example, to enable science data:

```
vsc.execute(Ssr(Enable));
```

and to disable the science data...

```
vsc.execute(Ssr(Disable));
```

3.3.6 Enabling Diagnostic Monitoring

The spacecraft has the responsibility to monitor ninety-two (92) different temperatures and voltage whose origin is the LAT. This information is acquired and brought to the ground as one component of the spacecraft's telemetry. In order to mitigate against single point failure, each quantity is brought to the spacecraft from the LAT *twice*, therefore, in actuality, this corresponds to monitoring *184* quantities. The spacecraft has block redundancy, with one set of signals (the *primary* signals) being monitored by one monitor and the other set (the *redundant* signals) being monitored by another. At any one time only a single set of signals are acquired and sent to the ground. As the VSC emulates the spacecraft it must provide analogous functionality. In addition, to support ground operations the VSC must also monitor 10 different voltages and currents from the BPU. Both these functions are provided by the so-called *monitor* interface. Physically, this information is brought from the LAT to the VSC

through the VSC's *850* and *468* boards (see Chapter 2). Once brought to the VSC these data are digitized by the VSC's *Systran* boards (see Chapter 2). The monitor software on the VSC periodically, once a second, reads these digitized temperatures and voltages, packages them up as a set of CCSDS telemetry packets and sends these packets down to the proxy through its VSC diagnostic stream (see Section 7.2). The monitor has two components: the *primary* monitor and the *redundant* monitor. The user must select which one of these two monitors will be used to collect this telemetry. Once this telemetry arrives at the proxy, the user handles this information in the same fashion as it would process any other telemetry (see Section 3.4). Monitor selection (primary or redundant) as well as its disabling is available through the proxy interface (see Section 7.10). When monitoring is disabled, the VSC no longer either acquires or transmits monitoring packets.

The VSC specifies one `APID` for the information digitized by the 850 board and one `APID` for the information digitized by the 468 board. Therefore, once a second, the user can expect to see two different packets on the diagnostic stream. The structure and `APID`s of these packets are defined in Appendix B. Note that the telemetry packet contains a field identifying whether data were acquired with the primary or redundant monitor. Different versions of the VSC can contain, *zero*, *one*, or *both* of these boards, consequently, the actual number of packets transmitted per second is dependent on the number of boards. For example, if a VSC contains only an 850 board, only the packet corresponding to that board is transmitted. If the VSC contains neither of these boards, nothing will be transmitted. In short, to acquire and process diagnostic telemetry from the proxy requires:

- at least one of either an 850 or 468 board
- the corresponding *Systran* boards
- cabling correctly established between LAT and VSC (and potentially a BPU)
- router and handler connected to VSC diagnostic stream
- monitoring to be enabled

In order to enable or disable monitoring one instantiates the `Monitor` class (see Section 7.10). The argument to its constructor is an enumeration specifying which of two monitors to enable. For example, to enable monitoring from the *Primary* monitor:

```
vsc.execute(Monitoring(Primary));
```

and to disable monitoring...

```
vsc.execute(Monitoring(None));
```

Finally, to enable monitoring from the *Redundant* monitor:

```
vsc.execute(Monitoring(Redundant));
```

*Note: When the VSC is initially booted, monitoring is **disabled**.*

3.4 Handling Incoming telemetry

In order to catch and process telemetry an application *sub-classes* from one of two handlers described in chapter 5:

- `TelemetryHandler`
- `ScienceHandler`

As an example, imagine the construction of a set of tools to assist a user in debugging an application which processes telemetry. One potentially useful tool would be a mechanism which simply decodes and prints the invariants associated with any CCSDS telemetry packet. The following code fragment captures the spirit of this functionality:

```
void decode(VscCcsds::Telemetry& packet, const char* title)
  {
  printf(title);
  printf("Apid %x and function %x\n", packet.apid(), packet.function());
  printf("Acquired at %d,%d\n", packet.secs(),packet.usecs());
  printf("Payload size is: %d bytes\n", packet.sizeofPayload());
  printf("Dump of first word of payload...\n", *packet.payload());
  return;
  };
```

This function is passed the packet to decode and print along with a character string to print as the decode prefix[1]. Now, in order to simplify the examples in Section 3.5.2, gratuitously wrap this function in a class which accepts the packet title as an argument to its constructor:

```
class Dump {
  public:
    Dump(const char* title) {_title = title}
   ~Dump() {}
  public:
    void decode(VscCcsds::Telemetry&);
  private:
     const char* _title;
  };
```

1. Of course, this function would, in practice, be much more sophisticated.

Next, a class, derived from `TelemetryHandler` (see Section 5.3) is defined which contains this function...

```
class MyHandler: public TelemetryHandler {
  public:
    MyHandler(const char* title) : TelemetryHandler(), _dump(title) {}
   ~MyHandler() {}
  private:
    Dump  _dump;
  };
```

`MyHandler` contains an object of type `Dump`. A handler disposes of its caught packets through its `process` method. The method is passed, as an argument, the arrived packet. As this method is virtual, `MyHandler` provides the implementation:

```
void MyHandler::process(VscCcsds::Telemetry& packet)
  {
  _decode(packet);
  return;
  }
```

The implementation of this method simply calls its own decode function. To actually catch and process arriving telemetry, this handler must be both instantiated and registered with a *router* as discussed in Section 3.5.

3.4.1 APID filtering

In many cases, a handler's goal is to catch a specific type of telemetry. Typically, the type of a CCSDS packet is determined by its `APID`. Therefore, catching a specific type of telemetry amounts to catching a specific set of `APID`s. This requires a straightforward usage of the methods of the `Handler` and `ApidRange` classes. The `ApidRange` class allows the user to define a *range* of `APID`s. The constructor for this class requires two short numbers. One number corresponds to the *lowest* `APID` of the range and the other the *highest* `APID` of the range. If the range corresponds to a single `APID`, the low and high values are *equal*. The `Handler` maintains a list of `ApidRange`s. This list defines the set of `APID`s which the handler is prepared to catch.

As an example, assume a handler is necessary to catch and processes *only* `LAT` housekeeping. As defined by the ICD (see [18]) all housekeeping `APID`s fit with a range which varies from `200` to `25F`, hexadecimal. The handler needed for housekeeping simply inherits from the previously defined `MyHandler`...

```
Houeskeeping : public MyHandler {
  public:
    Housekeeping();
   ~Housekeeping() {}
};
```

And the constructor's implementation...

```
Housekeeping::Housekeeping() : MyHandler("Housekeeping...")
  {
  handle(0x200, 0x25F);
  }
```

The implementation simply calls `Router`'s `range` method, passing both the necessary low and high `APID`s. Once registered with a router, the handler will catch and process those packets whose `APID`s fall only within the housekeeping range. However, what happens to those packets which do *not* fall within this range specified by the handler? In such a case, these packets are delivered to a router's `catchall` method as discussed in Section 3.5. What if a handler wishes to catch more then one range? For example, assume (poorly), the `Housekeeping` handler is required to catch and process not only `LAT` housekeeping telemetry, but also alert telemetry. Alert `APID`s vary from `340` to `39F` hexadecimal. In such a case, the `range` method is used twice:

```
Housekeeping::Housekeeping() : MyHandler("Housekeeping...")
  {
  handle(0x200, 0x25F); // housekeeping...
  handle(0x340, 0x39F); // alert...
  }
```

3.5 Routing Incoming telemetry

Telemetry arrives from the VSC to the proxy interface as a series of CCSDS packets (see 4.7 and 4.8). As these packets arrive they need to be dispatched to a set of appropriate telemetry handlers (telemetry handlers are discussed in Section 3.4). In order to dispatch arriving telemetry, the application provides the proxy interface with a *router*. A router is an application specific class inheriting from `Router` (contained in the `Routing` package described in Chapter 6). Once instantiated, a router is registered with the proxy interface. Once registered, the router is invoked by the proxy interface for each arriving CCSDS packet.

There are four steps involved in constructing an application specific router:

i. Sub-class from the router base class.

ii. Implement memory management strategy for arriving packets. If the processing of the packet can take processed entirely within the context of the router, memory for only one packet is necessary. If the processing of the packet is delegated to other processes, perhaps executing in a different thread, then a scheme where packets are allocated and deallocated from free-store may be necessary.

iii. Specify the set of necessary packet handlers.

iv. Implement a "catchall". This is a handler to designed to catch any packets which were not caught by the specified set of packet handlers.

As an example, suppose a router is specified which process packets entirely in its own context. This implies a fixed packet allocation scheme is sufficient. If this router's name is "`MyRouter`", its class definition would be as follows:

```
class MyRouter: public TelemetryRouter {
  public:
    MyRouter()
   ~MyRouter() {}
  private:
    Dump                _dump;
    VscCcsds::Telemetry _packet; // one packet's worth of storage...
  };
```

`MyRouter` now contains an object of type `Dump` (used for the `catchall` method, see below) and also enough storage for any one telemetry sized packet. The constructor's implementation simply registers the housekeeping handler defined in Section 3.4...

```
MyRouter::MyRouter() : _dump("Unexpected packet...")
  {
  insert(*(new Houskekeeping));
  }
```

When a packet arrives, it will be copied into this storage by the proxy interface. The interface locates *where* to copy the packet by invoking the router's `allocate` method. As this method is virtual, `MyRouter` provides the implementation, which is simply...

```
VscCcsds::Telemetry& MyRouter::allocate() {return _packet;}
```

If the handler cannot catch all packets, they will be caught and processed by the router's `catchall` method. This method is virtual, so `MyRouter must` provides the implementation...

```
void MyRouter::catchall(Telemetry& packet)
  {
  _dump.decode(packet);
  return;
  }
```

Note, that the `catchall` handler has the same call interface as a handler's `process` member. Indeed, in this example, the implementation almost matches `MyHandlers`'s `process` implementation. The only difference being the printing of the header string proceeding the `decode` function. To actually dispatch arriving telemetry, the router must be both instantiated and registered. This process is described in the next section.

3.5.1 Router registration

Telemetry is delivered by the proxy interface on *five* asynchronous streams. That implies telemetry on any one stream arrives both unsolicited and independently with respect to telemetry on any other stream. The proxy interface dispatches telemetry to an application through a router (see Section 3.5). The application creates the router and registers the router with the proxy by calling an appropriate method of the `Proxy` class (see Section 7.2). The correspondence between stream and registration method is enumerated in Table 7. For example, to catch telemetry on the LAT telemetry stream, the application would register their router by invoking the `latTelemetry` function of the `Proxy` class, passing to the function, as an argument, the router to be registered.

Table 7 Streams and registration methods

Stream	Registration Method
VSC diagnostic	`vscDiagnostic`
LAT diagnostic	`latDiagnostic`
VSC telemetry	`vscTelemetry`
LAT telemetry	`latTelemetry`
Science telemetry	`latScience`

The proxy interface associates a *thread* with each stream. The execution code represented by an application's router (and its set of handlers) is all executed in the context of that thread. The relative priority between the five potential threads is fixed by the proxy interface and cannot be changed. These relative priorities are implicit in the ordering of Table 7. For example, the highest priority thread is associated with the VSC diagnostic stream, while the lowest is associated with the science stream. While relative priorities are immutable, the *base* priority of the five threads can be changed by the application (see Section 7.14). This should only prove necessary in the eventuality that the proxy interface's threads conflict with the priorities of other application threads.

As an example, suppose the router described in Section 3.5 was required catch and process arriving LAT telemetry. This requires the router to first be instantiated and second, passed to the appropriate member function to be registered...

```
vsc.latTelemetry(*(new MyRouter));
```

3.5.2 APID dispatching

The previous section demonstrated how an application constructs a router which dispatches all packets belonging to a single handler. However, the interface also supports dispatching packets (based on APID) to different handlers. In the same fashion that a handler contains a list of APIDs, a router contains a list of handlers. For example, assume we wish to process alert telemetry differently then housekeeping telemetry. Start by reusing the HoueseKeeping class[1]. Next, define a new handler called "Alert"...

```
class Alert: public MyHandler {
  public:
    Alert()
   ~Alert() {}
  };
```

For clarity, Alert has almost the exact implementation as Houeskeeping. In substance it differs only in its constructor implementation...

```
Alert::Alert() : MyHandler("Alert...")
  {
  range(0x340, 0x39F);
  }
```

with the implementation varying in the alert APID range and title string. A real example, would, of course, take a very different form. Finally, modify MyRouter's constructor and register *two* handlers...

```
MyRouter::MyRouter() : _dump("Unexpected packet...")
  {
  insert(*(new Alert));
  insert(*(new Houskekeeping));
  }
```

1. But in this case we limit the handler to process *only* housekeeping APIDs.

3.5.3 Science telemetry and datagrams

To be written. At this point there are still some ambiguities about the structure of the science data which I'm trying to resolve. Stay tuned.

3.6 Telecommands

Control of the LAT through the VSC is exercised by queuing telecommands to the VSC. Telecommands are CCSDS packets which include a GLAST specific secondary header (see [8]). This structure is captured by the `TeleCmnd` class (described in Section 4.4) which is part of the data handling package. While all telecommands share the same shape, they all differ in their respective payloads. This means in practice, for most applications, specific telecommands are constructed by derivation using `TeleCmnd` as a base class. The customization of these classes typically will involve additional constructors and access methods. For example, consider an implementation of the LAT specific telcommand `SysReset` (see [18]). This command resets the LAT's housekeeping system using a specified set of configuration files. The parameters for this telecommand packet are as follows:

— The `APID` is `650` (hexadecimal)

— The function code is one (1)

— The payload contains one four (4) element array. Each element of the array is an unsigned 32-bit value which specifies a configuration file ID.

These parameters can be summarized with two `#defines` and one structure...

```
struct _MyPayload {unsigned fileIds[4];};

#define APID     0x0650
#define FUNCTION 1
```

First, the class definition, which is named after the telecommand mnemonic. The IDs for configurations files will be specified as arguments to the constructor. The number of configuration files can actually vary from zero to four, which implies four constructors. For pedagogical reasons, only the first few constructors are shown in the definition. In order to access the file IDs the class has four methods, the name of the method corresponding to the corresponding file ID passed in the constructor. The definition for this class...

```
class SysReset : public TeleCmnd {
  {
  public: // constructors...
    SysReset(unsigned fid0 = 0, unsigned fid1 = 0,...);
  public: // destructor...
   ~SysReset();
  public:
    unsigned fid0() const;
    unsigned fid1() const;
    unsigned fid2() const;
    unsigned fid3() const;
  };
```

Although the documentation is not explicit with respect to unused entries in the file ID array, the constructors will initialize these entries to *null*. As an example, the following code fragment is the implementation of the constructor which requires *two* IDs:

```
SysReset::SysReset(unsigned fid0, unsigned fid1) : TeleCmnd(APID, FUNCTION)
  {
  _MyPayload payload;

  payload.fileId[0] = fid0;
  payload.fileId[1] = fid1;
  payload.fileId[2] = 0;
  payload.fileId[3] = 0;

  copy((void*payload, sizeof(_MyPayLoad));
  };
```

Last, the implementation of one of the four accessor methods:

```
unsigned SysReset::fid2()
  {
  unsigned long* fileId = (unsigned long*)payload();

  return fileId[2];
  };
```

3.7 Scheduling the “Magic seven”

Once a second, the VSC sends to the LAT *seven* (7) ancillary data telecommands. These are the so-called “magic seven” commands (see Section 4.9). Five of these telecommands are identical in structure and contain *attitude* information (Section 4.10), one contains miscellaneous data

(Section 4.11), and one contains the *timetone* for the subsequent 1-PPS signal (see Section 4.12). Within each one second period, these commands are sequenced and scheduled in a predefined order, as specified in [8] and discussed in Section 1.4.3. The pertinent details for ancillary sequencing are as follows:

— The scheduler divides each one second period into twenty-five fixed time slots of *40000* microseconds each.

— As the spacecraft limits transmission to at most one command per slot, only one of the magic seven is sent within any one slot.

— The ICD calls out a specific time slot for each of the magic seven commands.

If one assumes that t_0 represents the VSC's initial observatory time-base (see Section 1.3), Figure 22 illustrates the scheduling process:

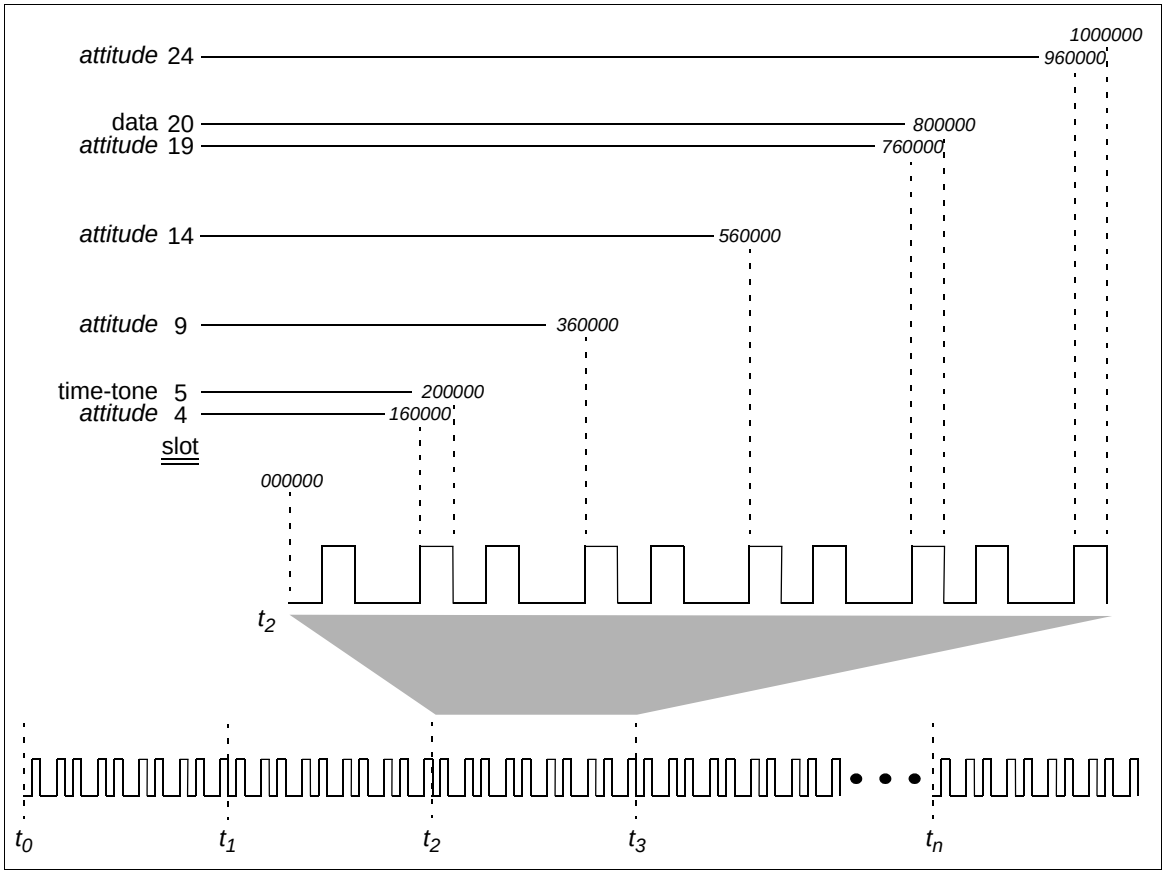

Figure 22 Scheduling the "Magic seven"

For example, when requested, the timetone command is always transmitted by the VSC somewhere within slot *five*, while the 4th attitude command is always sent somewhere within slot *nineteen*. Slot number 5 spans the interval from *200000* to *239999* microseconds, while slot number 19 spans the interval from *760000* to *799999* microseconds.

The proxy interface provides the application with three different options in terms of how the ancillary slots are populated:

— Do *not* fill. This is the default behaviour after the VSC boots. Of course, this option is not very useful with respect to correct operation of the LAT and is intended primarily as a tool to commission and debug the processing of the ancillary sequence by FSW on the LAT. If a default sequence (see below) has been established, one can return to not filling the ancillary slots by calling the `schedule` member of the `Proxy` class with *no* arguments.

— Fill with the *same* set of commands, independent of period. The application defines a specific set of seven commands, sends these commands to the VSC, which transmits the same set at every period to the LAT. This option is attractive when LAT FSW either needs to dwell on a specific set, or doesn't care what the contents of the commands are, as long as the appropriate slots are filled. The application specifies a fixed sequence by calling the `schedule` member of the `Proxy` class, passing as arguments the seven ancillary telecommands (see Section 7.2) to be used as the default set.

— Fill with a *different* set of commands for each period. The application defines a set of commands, queues these commands to the VSC where they are stored waiting execution on the appropriate period. Once a sequence for a particular period is defined, the application calls the `schedule` member of the `Proxy` class (see Section 7.2) to queue the sequence for that period. The arguments to this function specific the seven commands of the sequence along with the particular time period the corresponding commands should be scheduled. A period is specified in GLAST standard units (the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001).

For example, this code snippet establishes a default sequence[1]...

```
vsc.schedule(Attitude(170000, ...),
             TimeTone(210000, ...),
             Attitude(370000, ...),
             Attitude(570000, ...),
             Attitude(770000, ...),
             Data(810000, ...),
             Attitude(970000, ...));
```

The first argument of any magic seven command specifies the acquisition time of the data associated with that instance of the command. The acquisition time is specified in *microseconds* relative to the time period. For example, the code snippet above specifies that the data for the timetone command always occurs `2100` microseconds into any one period.

Note, that the LAT constrains the acquisition time for a command to fall within the slot corresponding to that command. If this constraint is violated the LAT issues an error when processing a ancillary command. By specifying an invalid acquisition time an application can test whether this constraint is correctly enforced. For example....

1. For clarity, the bulk of the construction arguments are omitted.

```
vsc.schedule(Attitude(160000, ...),   // OK...
             TimeTone(239900, ...),   // OK...
             Attitude(380000, ...),   // OK...
             Attitude(660000, ...),   // Error...
             Attitude(762100, ...),   // OK,,,
             Data(801000, ...),       // OK...
             Attitude(1020000, ...)); // Error...
```

To return to the VSC to its default state of *not* filling the ancillary slots requires scheduling a null sequence...

```
vsc.schedule();
```

Finally, using Figure 22 as an example, to schedule a sequence for the second period relative to the initial value of the observatory time-base (t_0)...

```
#define T2 (T0 + (unsigned)2) // T0 is the initial observatory base time...

vsc.schedule(T2, Attitude(...),
                 TimeTone(...),
                 Attitude(...),
                 Attitude(...),
                 Attitude(...),
                 Data(...),
                 Attitude(...));
```

3.8 Administrating the VSC

3.8.1 Configuring queue sizes

Each of the six queues described in Section 1.4.1 is assigned, by default a maximum number of entries (see Table 4). In many cases, the default values may prove insufficient. In such a case, the user may change queue sizes by manipulating an argument to the `init_vsc` procedure. This procedure is called within the VSC's startup script. This script is managed as part of any VSC release and consequently is under the control of the FSW code management and release system (see xxx). The path specification for this script is as follows:

```
"VSC/<tag>/ptd/startup-mv2304-vsc.vx"
```

where the `<tag>` token is replaced with a number corresponding to whatever release is appropriate to the user. The argument used to modify the default values is a string which

consists of a comma separated list, each item of the list corresponding to the value for a particular queue. The syntax of this string is as follows:

```
"<name>:<count>, <name:<count>, ..."
```

Tokens expressed in brackets (for example, <name>) are replaced by the user. There are two replaceable tokens:

"name" Is a string corresponding to the name of the queue. Allowed queue names are enumerated in Table 4. Correct capitalization is necessary.

"count" Is an unsigned, *non-zero* number corresponding to the size (in entries) of the queue. The number's radix can be expressed in either decimal or hexadecimal. A hexidecimal value is specified by the digit `0`, followed by one of the letters `x` or `X` followed by a string of hexadecimal digits. The hexadecimal digits are the digits `0` through `9`, plus the characters `a` through `f` (or `A` through `F`), which have the values 10 through 15, respectively. For example, a queue size of twenty-nine could be expressed as either `29`, `0X1D`, or `0x1d`.

For example, to change the size of the `Ancillary` queue the signature of the `init_vsc` procedure could be as follows:

```
init_vsc("Ancillary:1000000");
```

This would create an `Ancillary` queue capable of buffering up to one million (decimal) seconds[1] worth of ancillary sequences and the other five queues would retain their default values. If, one the other hand, two queues required new values...

```
init_vsc("Ancillary:1000000, CommandStored:0x1F");
```

3.8.2 Network configuration

3.8.2.1 VSC node name and IP address

To be written.

1. One entry on the ancillary queue holds all seven ancillary commands.

3.8.2.2 Port Numbers

To be written.

Chapter 4
CCSDS package

The principal quanta of information between the VSC and the LAT and consequently the principal quanta of information between VSC and application is the CCSDS packet (see [16] and [17]). This package contains the necessary class support for construction of these packets. GLAST specifies two different types of CCSDS packets: *Telecommands* and *Generic Telemetry* (see [8]). This package provides class representations for both types: `TeleCmnd` (see Section 4.4) and `GenericTelemetry` (see Section 4.6). These representations derive from a common, abstract representation of a CCSDS packet. The LAT specifies that telemetry comes in two forms: `Telemetry` and `Science`. Applications are expected to construct specific commands and telemetry by sub-classing from either of these two classes. The package itself sub-classes from `TeleCmnd` in order to form the three different types of telecommands used in the definition of a "Magic seven" sequence. The usage of these specific type of commands is discussed in Section 4.9. The dependencies for the classes of this package are illustrated in Figure 23:

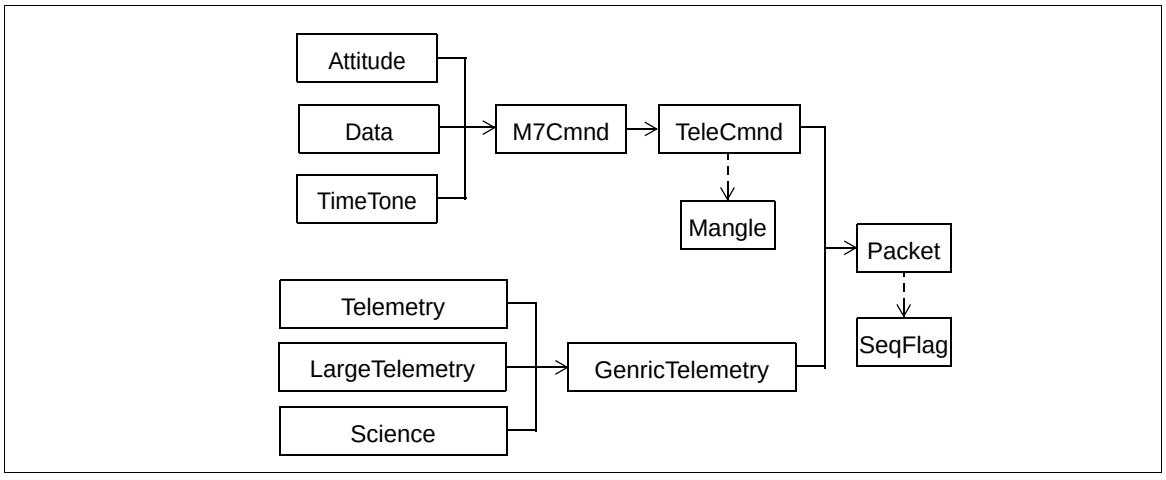

Figure 23 Class dependencies for the CCSDS package

4.1 Name space - `VscCcsds`

4.2 Packet sequencing

Sequencing is used when any one piece of information transmitted, using CCSDS packets, cannot fit within the payload of any *one* packet. In other words, the information *spans* packets. Packet sequencing is not applicable for GLAST telecommands as they are fixed size (64 bytes) and will always fit within a single packet. However, this is not necessarily true for GLAST telemetry. In such a case, the sequence flags and sequence number will, working in conjunction, identify what position in a sequence of information, the packet occupies. This position information is enumerated within Table 8:

Table 8 Enumeration of sequence flag for a CCSDS telemetry packet

Enumeration	Value	The packet is...
`middle`	`0`	part of a sequence, but neither the first nor the last packet in the sequence
`first`	`1`	the *first* packet of a sequence.
`last`	`2`	the *last* packet of a sequence.
`alone`	`3`	*not* part of a sequence.

These values are captured in the following enumeration:

Listing 1 Enumeration for packet sequence flags

```
1:  enum SeqFlags {middle=0, first=1, last = 2, alone = 3};
```

4.3 Packet

This class represents a generic CCSDS packet, independent of whether the packet contains either telemetry or a telecommand. An application never directly inherits from the class. Instead, this class is used indirectly, through inheritance, via the `TeleCmnd` and `Telemetry` classes (see Section 4.4 and Section 4.6). Of course, the class's member functions still interest an application. These methods fit into three generic categories:

— Header Inspectors. Those methods returning the primary header attributes of the packet. For example, the `version` member.

— Payload Management. Those methods which both manage and control the user payload of the packet. For example, the `payload` member.

— Sequencing. Those methods which are used to control packet sequencing. For example, the `seqFlags` function.

The derived classes's interest will most likely be centered around the payload methods as these are the primary mechanism used by an application in order to provide specialization of user's telecommands and telemetry.

Listing 2 Class definition for `Packet`

```
 1: template<unsigned T> class Packet {
 2:   protected: // constructors...
 3:     Packet(short apid);
 4:     Packet(short apid, Ccsds::SeqFlags, unsigned short seqNum);
 5:     Packet(const Packet&);
 6:   public:
 7:    ~Packet();
 8:   public:
 9:     Packet& operator = (const Packet&);
10:   public: // inspectors...
11:     unsigned short  apid()          const;
12:     unsigned short  version()       const;
13:     unsigned short  length()        const;
14:     unsigned short  seqNum()        const;
15:     Ccsds::SeqFlags seqFlags()      const;
16:     bool            isTelecommand() const;
17:   };
```

4.3.1 Constructor synopsis

`Packet(short apid)` This constructor initializes the CCSDS primary header of the packet corresponding to the object as follows:

- the `APID` is set to the value of the constructor's first argument.
- the secondary header (`SH`) is *set*.
- the packet type identifier (`T`) is *set*, indicating this is a *telecommand*.
- the packet version is set to *zero*, indicating a version 1 packet.
- the sequence count is set to *zero*.
- the sequence flag is set to "Standalone" packet (3).
- the packet length is set to *zero* (0).

The constructor throws *no* exceptions.

`Packet(short apid, SeqFlags, unsigned short seqNum)` This constructor initializes the CCSDS primary header of the packet corresponding to the object as follows:

- the `APID` is set to the value of the constructor's *first* argument.
- the secondary header (`SH`) is *set*.
- the packet type identifier (`T`) is *cleared*, indicating this is a *telemetry* packet.
- the packet version is set to *zero*, indicating a version 1 packet.

— the sequence count is set to the value specified by the constructor's *third* argument. If this argument is omitted, the sequence count is set to *zero*.

— the sequence flag is set to the value specified by the constructor's *second* argument.

— the packet length is set to *zero* (0).

The constructor throws *no* exceptions.

4.3.2 Member synopsis

apid Returns a value corresponding to the application identifier (APID) for the corresponding CCSDS packet. APIDs range from *zero* (0) to FFFF (hex). This function has no arguments and throws no exceptions.

isTelecommand Returns an boolean identifying whether or not the corresponding CCSDS packet is a command or telemetry packet. If the value returned is TRUE, the packet is a *telecommand* packet. If the value returned is FALSE, the packet is a *telemetry* packet. This function has no arguments and throws no exceptions.

version Returns a value corresponding to the version identifier for the corresponding CCSDS packet. The version identifier for all LAT CCSDS packets is *zero* (0), used to indicate a *Version 1* packet. This function has no arguments and throws no exceptions.

seqNum Returns a value corresponding to the sequence number for the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

seqFlags Returns an enumeration specifying the packet's location in a sequence of data which could potentially span multiple packets. The return value can have one of the four possible values enumerated within Table 8. This function has no arguments and throws no exceptions.

length Returns the length of the packet *less* the primary header (six bytes). For reasons which escape me, the value is actually the value specified above *less* one (1). The length is expressed in units of bytes. The length is inclusive of any secondary header. As all LAT CCSDS packets are a multiple of 16-bits words, the value returned will always be odd. This function has no arguments and throws no exceptions.

4.4 TeleCmnd

This class specifies a GLAST specific, CCSDS *telecommand* packet. All GLAST telecommands share a common set of attributes. In particular:

— the isTelecommand function always returns TRUE.

— The sequence flag field is always set to alone.

— telecommands have a fixed size (which is 64 bytes), independent of the number of significant bytes in the payload. That is, the `maxPayload` function will return 64 bytes less the header.

— A payload checksum. This checksum is always located at the end of the allocated payload. Each time the payload is copied, the checksum is re-computed.

A example of how an application constructs their own telecommands is found in Section 3.6.

Listing 3 Class definition for `TeleCmnd`

```
 1:  class TeleCmnd : public Packet<64> {
 2:    public: // constructors...
 3:      TeleCmnd(unsigned short apid = 0, unsigned short function = 0);
 4:      TeleCmnd(const Mangle&, short apid = 0, short function = 0);
 5:      TeleCmnd(const TelCmnd&);
 6:    public:
 7:     ~TeleCmnd();
 8:    public:
 9:      TeleCmnd& operator = (const TeleCmnd&);
10:    public:
11:      void copy(const void*, int sizeofPayload);
12:    public:
13:      unsigned short        function()      const;
14:      unsigned short        maxPayload()    const
15:      const unsigned short* payload()       const;
16:    public:
17:      unsigned short sizeofPayload()    const;
18:      unsigned short recordedChecksum() const;
19:      unsigned short computedChecksum() const;
20:    };
```

4.4.1 Constructor synopsis

`TeleCmnd(short apid, short function)` The CCSDS header is initialized in manner appropriate for a CCSDS telecommand packet (see [16] and [17]). The `APID` of the packet is derived from the *first* argument passed to the constructor and the packet's `function` code is derived from its *second* argument. The payload size is set to *zero*. The constructor throws *no* exceptions.

`TeleCmnd(const Mangle&, short apid, short function)` The CCSDS header is initialized in manner *inappropriate* for a CCSDS telecommand packet. This constructor should be used *only* when one is testing the LAT's response to an ill-formed telecommand. The *first* argument is a reference to an object (see Section 4.5) which defines which fields of the constructed packet should be malformed. The `APID` of the packet is derived from the *second* argument passed to the constructor and the packet's `function` code is derived from its *third* argument. The payload size is set to *zero*. The constructor throws *no* exceptions.

4.4.2 Member synopsis

`copy` Copies the data specified by the argument into the payload of the packet. Any data currently in the payload is lost. The first argument is a pointer to a buffer representing the data to be copied into the packet as the payload. The second argument specifies the size of the payload data (in bytes). The size must be an even number of bytes. The value of this argument must range from *two* (2) to the value returned by the `maxPayload` function described below. This function returns no value and throws no exceptions.

`function` Returns the telcommand's function code. This member function has no arguments and throws *no* exceptions.

`maxPayload` Returns the actual amount of storage reserved for a payload. The current allocation of the payload is returned by the function described above. This function has no arguments and throws no exceptions.

`payload` Returns a pointer to the payload portion of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

`sizeofPayload` Returns the size (in bytes) of the payload portion of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

`recordedChecksum` Returns the *recorded* payload check-sum for the packet. The `computedChecksum` function described below returns the *computed* checksum. A properly formed packet has a computed checksum which is equal to its recorded checksum. The recorded checksum is replaced each time the `resizePayload` function is called. This function has no arguments and throws *no* exceptions.

`computedChecksum` Returns the *computed* payload check-sum for the packet. The `recordedChecksum` function described above returns the *recorded* checksum. A properly formed packet has a computed checksum which is equal to its recorded checksum. This function has no arguments and throws *no* exceptions.

4.5 Mangle

This class is used to *consciously* mangle the format and structure of a GLAST telecommand. Nominally, of course, there would seem to be no need to purposely construct and transmit an ill-formed telecommand. The express reason for this class's existence is to *test* the LAT's response to invalid telecommands. When, for a particular application, this reason does not exist, its safe to assume the application can afford to ignore this class.

The class consists of a number of virtual functions, each function corresponding to a field of a telecommand which may be mangled. Each function returns a mangled value for its corresponding field. Because the functions are virtual, they may be over-loaded, allowing application to customize both the fields which should be mangled and their respective values. In practice, mangling is accomplished by instantiating either the class (or a class derived from it) and then passing the resulting object as an argument to `TeleCmnd`'s constructor (see

Section 4.4). Note, that by default (that is, using the base class without derivation), each member returns a value guaranteed to be inappropriate for its corresponding field.

Listing 4 Class definition for `Mangle`

```
 1: class Mangle {
 2:   public: // constructors...
 3:     Mangle();
 4:   public:
 5:     virtual ~Mangle();
 6:   public:
 7:     virtual bool  T() const;
 8:     virtual bool SH() const;
 9:   public:
10:     virtual unsigned short version()          const;
11:     virtual unsigned short recordedChecksum() const;
12:   };
```

4.5.1 Constructor synopsis

Just the default.

4.5.2 Member synopsis

`T` Returns the requested state of the "T" field of the CCSDS packet header (see [16] and [17]). The "T" field determines whether or not the packet declares itself as a telecommand. A value of `TRUE` specifies the "T" field is *set*, a `FALSE` value that the field is *clear*. Note, that this function is *virtual* and may be over-loaded by a derived class. By default (the member function is *not* over-loaded) the function returns a `FALSE` value. This function has no arguments and throws no exceptions.

`SH` Returns the requested state of the "SH" field of the CCSDS packet header (see [16] and [17]). The "SH" field determines whether or not the packet has a secondary header. A value of `TRUE` specifies the "SH" field is *set*, a `FALSE` value that the field is *clear*. Note, that this function is *virtual* and may be over-loaded by a derived class. By default (the member function is *not* over-loaded) the function returns a `FALSE` value. This function has no arguments and throws no exceptions.

`version` Returns the requested value of the "VERSION" field of the CCSDS packet header (see [16] and [17]). Only the low-order three bits of the returned value are significant. Note, that this function is *virtual* and may be over-loaded by a derived class. By default (the member function is *not* over-loaded) the function returns a value of *seven* (7). This function has no arguments and throws no exceptions.

`recordedChecksum` Returns the requested value for the *initial* value of the packet's recorded checksum. Note, that as soon as the `TeleCmnd`'s `copy` method (see Section 4.4) is called, this value will no longer be the packet's recorded checksum. Instead, the packet will have the computed checksum. Note, that this function is *virtual* and may be over-loaded by a derived class. By default (the member function is *not* over-loaded) the function returns a value of *zero* (0). This function has no arguments and throws no exceptions.

4.6 Generic Telemetry packet

This class specifies a GLAST specific, generic CCSDS *telemetry* packet. All GLAST telemetry packets share a common set of attributes. In particular:

— the `isTelecommand` function always returns `FALSE`.

— A time-stamp. This time-stamp specifies when the telemetry was acquired. Acquisition time is represented by two 32-bit quantities, the acquisition time in granularity of both seconds and micro-seconds. The seconds quantity represents the acquisition time as the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The micro-seconds quantity is the acquisition time in micro-seconds, relative to the seconds quantity.

LAT telemetry comes in two varieties, largely differentiated by their maximum size:

i. *Telemetry*: All small telemetry is fixed size, independent of actual allocated payload size (1024 bytes). That is, the `maxPayload` function will return 1024 bytes less the size of the header. Telemetry packets are delivered on the LAT and VSC telemetry streams (see Section 7.2).

ii. *Science*: All science telemetry is fixed size, independent of actual allocated payload size (4096 bytes). That is, the `maxPayload` function will return 4096 bytes, less the size of the header. Science packets are delivered on the LAT's science stream (see Section 7.2).

A discussion on how telemetry is processed by a user application is found in Section 3.4.

Listing 5 Class definition for `GenericTelemetry`

```
 1: class GenericTelemetry : public Packet<T> {
 2:   protected: // constructors...
 3:     GenericTelemetry(short apid,
 4:                      unsigned secs, unsigned usecs,
 5:                      SeqFlags, unsigned short seqNum);
 6:     GenericTelemetry(const GenericTelemetry&);
 7:   public:
 8:     virtual ~GenericTelemetry();
 9:   public:
10:     GenericTelemetry& operator= (const GenericTelemetry&);
11:   public:
12:     void copy(const void*, int sizeofPayload);
13:     void append(const void*, int sizeofBuffer);
14:   public:
15:     unsigned short        sizeofPayload() const;
16:     unsigned short        maxPayload()    const;
17:     const unsigned short* payload();      const;
18:   public:
19:     unsigned secs()  const;
20:     unsigned usecs() const;
21:   };
```

4.6.1 Constructor synopsis

`GenericTelemetry(short apid, unsigned secs, unsigned usecs, SeqFlags, short seqNum)` The CCSDS header is initialized in manner appropriate for a CCSDS telemetry packet (see [8]). The `APID` of the packet is derived from the first argument passed to the constructor. The allocated payload size is set to *zero*. The second and third arguments specify the time at which the telemetry payload was acquired. The second argument corresponds to the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The third argument corresponds to the acquisition time relative to the second argument. This time is measured in *micro*-seconds. The forth argument determines the position of the packet in a set of information which spans packets. If this argument is omitted, the sequence flag is set to *alone*. The fifth argument corresponds to the packet's sequence number. If this argument is omitted, the sequence number is set to *zero*. The constructor throws *no* exceptions.

4.6.2 Member synopsis

`copy` Copies the data specified by the argument into the payload of the packet. Any data currently in the payload is lost. The first argument is a pointer to a buffer representing the data to be copied into the packet as the payload. The second argument specifies the size of the payload data (in bytes). The size must be an even number of bytes. The value of this argument must range from *two* (2) to the value returned by the `maxPayload` function described below. This function returns no value and throws no exceptions.

`append` Appends the data specified by the argument into the payload of the packet. The first argument is a pointer to a buffer representing the data to be appended to the current payload. The second argument specifies the size of the appended data (in bytes). The size must be an even number of bytes. The value of this argument must be no larger then the difference between the values returned by the `sizeofPayload` and the `maxPayload` functions. If so, the remainder is simply not copied. This function returns no value and throws no exceptions.

`payload` Returns a pointer to the payload portion of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

`sizeofPayload` Returns the size (in bytes) of the payload portion of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

`maxPayload` Returns the actual amount of storage reserved for a payload. The current allocation of the payload is returned by the function described above. This function has no arguments and throws no exceptions.

`secs` Returns the time at which the telemetry payload was acquired. This time is returned as the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. This function has no arguments and throws no exceptions.

`usecs` Returns the time in *micro-seconds* relative to the time returned by the `secs` member function described above. This function has no arguments and throws no exceptions.

4.7 Telemetry

Telemetry packets are simply sub-classed from `GenericTelemetry` (see Section 4.6) with the template (packet size) argument satisfied. Telemetry packets are fixed size, independent of the actual allocated payload size (1024 bytes). That is, the `maxPayload` function will return 1024 bytes less the size of the header. Telemetry packets are delivered on both the LAT and VSC telemetry streams (see Section 7.2).

Listing 6 Class definition for `Telemetry`

```
 1:  class Telemetry : public GenericTelemetry<1024> {
 2:    public: // constructors...
 3:      Telemetry(short apid,
 4:                unsigned secs, unsigned usecs,
 5:                SeqFlags flags=alone, unsigned short seqNum=0);
 6:      Telemetry(const Telemetry&);
 7:    public:
 8:      virtual ~Telemetry();
 9:    public:
10:      Telemetry& operator= (const Telemetry&);
11:    };
```

4.8 Science

Science packets are simply sub-classed from `GenericTelemetry` (see Section 4.6) with the template (packet size) argument satisfied. Science packets are fixed size, independent of the actual allocated payload size (4096 bytes). That is, the `maxPayload` function will return 4096 bytes less the size of the header. Science packets are delivered on the science stream (see Section 7.2).

Listing 7 Class definition for `Science`

```
 1:  class Science : public GenericTelemetry<4096> {
 2:    public: // constructors...
 3:      Science(short apid,
 4:              unsigned secs, unsigned usecs,
 5:              SeqFlags flags=alone, unsigned short seqNum=0);
 6:      Science(const Science&);
 7:    public:
 8:      virtual ~Science();
 9:    public:
10:      Science& operator= (const Science&);
11:    public:
12:      unsigned short  pad()         const;
13:      const unsigned* data()        const;
14:      unsigned        sizeofData()  const;
15:    };
```

4.8.1 Member synopsis

`pad` Returns the value of the so-called “pad” word of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

`data` Returns a pointer to the datagram portion of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

`sizeofData` Returns the size (in bytes) of the datagram portion of the corresponding CCSDS packet. This function has no arguments and throws no exceptions.

4.9 The “Magic 7” Telecommands

To be written.

Listing 8 Class definition for `M7Cmnd`

```
 1:  class M7Cmnd : public TeleCmnd {
 2:    public:
 3:      enum FunctionCode {Attitute=1, Data=2, TimeTone=3};
 4:    public: // constructors...
 5:      M7Cmnd(FunctionCode);
 6:      M7Cmnd(const M7Cmnd&);
 7:    public:
 8:      ~M7Command();
 9:    public:
10:      M7Cmnd& operator = (const M7Cmnd&);
11:    public:
12:      unsigned period() const;
13:      unsigned isPeriod(unsigned time);
14:    };
```

4.9.1 Constructor synopsis

`M7Cmnd` This constructor builds a generic *ancillary* GLAST telecommand as specified in [8]. This class forms the base class for the specific ancillary telecommands described in sections 4.10, 4.10, and 4.10 and thus has little or no intrinsic interest in and of itself. The argument corresponds to one of the three possible functions codes corresponding to the three different types of ancillary telecommands. The execution *period* time-field associated with the ancillary telecommand is initialized to *zero* (0). Ancillary telecommands are transmitted for execution by objects of the class described in Section 7.2.

4.9.2 Member synopsis

`period` This function returns the current period value. This function throws *no* exceptions.

isPeriod Replaces the packet's current period value with the value specified by the argument. Period time is represented as the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. This function returns the previous period value. This function throws *no* exceptions.

4.10 The Attitude Ancillary Telecommand

To be written.

Listing 9 Class definition for `Attitude`

```
 1: class Attitude : public M7Cmnd {
 2:   public: // constructors...
 3:     Attitude(unsigned usecs,
 4:              double QSJ_1,
 5:              double QSJ_2,
 6:              double QSJ_3,
 7:              double QSJ_4,
 8:              float  WSJ_1,
 9:              float  WSJ_2,
10:              float  WSJ_3);
11:     Attitute& Attitude(const &Attitude);
12:   public: // destructor...
13:    ~Attitude();
14:   public:
15:     Attitude& operator = (const Attitude&);
16:   public:
17:     unsigned usecs() const;
18:   public:
19:     double QSJ_1() const;
20:     double QSJ_2() const;
21:     double QSJ_3() const;
22:     double QSJ_4() const;
23:     float  WSJ_1() const;
24:     float  WSJ_2() const;
25:     float  WSJ_3() const;
26:   };
```

4.10.1 Constructor synopsis

Attitude This constructor builds an attitude ancillary GLAST telecommand as specified in [8]. The first argument specifies the time *within* the execution period to schedule the telecommand. This time is specified in *micro-seconds*. and may range from a value of *zero* (0) to 999, 999 (decimal). The next four arguments are quaternions which correspond to the *attitude* of the Spacecraft, with the third argument corresponding to the attitude of the *X-axis*, the fourth the *Y-axis*, the fifth the

Z-axis, and the sixth, the so-called *scaler* component. The last three arguments correspond to the Spacecraft's *angular velocity*, with the seventh argument corresponding to the velocity of the *X-axis*, the eight the *Y-axis*, and the ninth the *Z-axis*. Ancillary telecommands are transmitted for execution by objects of the class described in Section 7.2.

4.10.2 Member synopsis

`usecs` Returns the time *within* the execution period to schedule the telecommand. This time is specified in *micro-seconds*. and may range from a value of *zero* (0) to 999, 999 (decimal). This function has *no* arguments.

`QSJ_1` Returns the *X-axis* component of the quaternion expressing the attitude of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes' constructor. This function has *no* arguments.

`QSJ_2` Returns the *Y-axis* component of the quaternion expressing the attitude of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes' constructor. This function has *no* arguments.

`QSJ_3` Returns the *Z-axis* component of the quaternion expressing the attitude of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes' constructor. This function has *no* arguments.

`QSJ_4` Returns the *scaler* component of the quaternion expressing the attitude of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes' constructor. This function has *no* arguments.

`WSJ_1` Returns the *X-axis* component expressing the Spacecraft's (S) angular velocity. Angular velocity is expressed in radians/second. This value was specified as an argument to the classes' constructor. This function has *no* arguments.

`WSJ_2` Returns the *Y-axis* component expressing the Spacecraft's (S) angular velocity. Angular velocity is expressed in radians/second. This value was specified as an argument to the classes' constructor. This function has *no* arguments.

`WSJ_3` Returns the *Z-axis* component expressing the Spacecraft's (S) angular velocity. Angular velocity is expressed in radians/second. This value was specified as an argument to the classes' constructor. This function has *no* arguments.

4.11 The Ancillary Data Telecommand

To be written.

Listing 10 Class definition for `Data`

```
 1:  class Data : public M7Cmnd {
 2:    public:
 3:      struct Flags {bool         LAT_IN_SSA  : 1;
 4:                    bool         IS_IN_SUN   : 1;
 5:                    bool         GPS_OUTAGE  : 1;
 6:                    bool         SBAND_ON    : 1;
 7:                    bool         XBAND_ON    : 1;
 8:                    bool         GBM_IN_SSA  : 1;
 9:                    bool         ARR_ENABLED : 1;
10:                    unsigned int RESERVED    : 9};
11:
12:      enum GncMode {Idle                    = 0,
13:                    Intertial_Capture      = 1,
14:                    Sun_Point              = 2,
15:                    Mission_Interial_Point = 3,
16:                    Mission_Manuever       = 4,
17:                    Mission_Zenith_Point   = 5,
18:                    Reentry_Cruise         = 6,
19:                    Reentry_Delta_V        = 7};
20:    public: // constructors...
21:      Data(unsigned usecs,
22:           float POSITION_X,
23:           float POSITION_Y,
24:           float POSITION_Z,
25:           float VELOCITY_X,
26:           float VELOCITY_Y,
27:           float VELOCITY_Z,
28:           Data::Gnc_Mode,
29:           unsigned SSR_USAGE,
30:           Data::Flags);
31:      Data& Data(const &Data);
32:    public: // destructor...
33:     ~Data();
34:    public:
35:      Data& operator = (const Data&);
36:    public:
37:      unsigned usecs() const;
38:    public:
39:      float          POSITION_X() const;
40:      float          POSITION_Y() const;
41:      float          POSITION_Z() const;
42:      float          VELOCITY_X() const;
43:      float          VELOCITY_Y() const;
44:      float          VELOCITY_Z() const;
45:      Data::GncMode  GNC_MODE()   const;
46:      unsigned       SSR_USAGE()  const;
47:      Data::Flags    FLAGS()      const;
48:    };
```

4.11.1 Constructor synopsis

`Data` This constructor builds an attitude ancillary GLAST telecommand as specified in [8]. The first argument specifies the time *within* the execution period which the information contained in the packet was acquired. This time is specified in *micro-seconds*. and may range from a value of *zero* (0) to 999, 999 (decimal). The next three arguments are quaternions which correspond to the *position* of the Spacecraft, with the first of the three arguments corresponding to the position with respect to the *X-axis*, the second with respect to the *Y-axis*, the third with respect to the *Z-axis*, The last three arguments correspond to the *velocity* of the spacecraft, with the first of the three arguments corresponding to the position with respect to the *X-axis*, the second with respect to the *Y-axis*, the third with respect to the *Z-axis*. The ninth argument is an enumeration corresponding to the satellite's GNC mode. The tenth argument specifies the fraction of SSR storage used for science data not yet down-linked. This fraction is expressed as a number between *zero* (0) and 100 percent. The last argument is a bit list representing the current state of the spacecraft. (See [7] for an explanation of the possible states). The Ancillary telecommands are transmitted for execution by objects of the class described in Section 7.2.

4.11.2 Member synopsis

`usecs` Returns the time *within* the execution period to schedule the telecommand. This time is specified in *micro-seconds*. and may range from a value of *zero* (0) to 999, 999 (decimal). This function has *no* arguments.

`POSITION_X` Returns the *X-axis* component of the quaternion expressing the *attitude* of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes's constructor. This method has *no* arguments and throws *no* exceptions.

`POSITION_Y` Returns the *Y-axis* component of the quaternion expressing the *attitude* of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes's constructor. This method has *no* arguments and throws *no* exceptions.

`POSITION_Z` Returns the *Z-axis* component of the quaternion expressing the *attitude* of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes's constructor. This method has *no* arguments and throws *no* exceptions.

`VELOCITY_X` Returns the *X-axis* component of the quaternion expressing the *velocity* of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes's constructor. This method has *no* arguments and throws *no* exceptions.

VELOCITY_Y Returns the *Y-axis* component of the quaternion expressing the *velocity* of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes's constructor. This method has *no* arguments and throws *no* exceptions.

VELOCITY_Z Returns the *Z-axis* component of the quaternion expressing the *velocity* of the Spacecraft (S) with respect to the J2000 frame (J). This value was specified as an argument to the classes's constructor. This method has *no* arguments and throws *no* exceptions.

GNC_MODE Returns an enumeration corresponding to the satellite's GNC mode. This method has *no* arguments and throws *no* exceptions.

SSR_USAGE Returns a value which specifies the fraction of SSR storage used for science data not yet down-linked. This fraction is expressed as a number between *zero* (0) and 100 percent. This method has *no* arguments and throws *no* exceptions.

FLAGS Returns a bit list representing the current state of the spacecraft. See [7] for an explanation of the possible states. This method has *no* arguments and throws *no* exceptions.

4.12 The Time-Tone Ancillary Telecommand

Objects of this class are instantiated automatically by the VSC in response to processing a queued ancillary sequence (see Section 7.2). This command indicates the exact time at the *next* transmission of the 1-PPS signal sent to the LAT (see Section 1.3). Nominally, this telecommand is sent to the LAT somewhere between 500 and 800 *milliseconds* before the corresponding 1-PPS signal is asserted. The LAT's timing system uses both this telecommand and 1-PPS signal to establish its absolute time in a fashion which is coherent with the Spacecraft's absolute time.

Listing 11 Class definition for `TimeTone`

```
 1:  class TimeTone : public M7Cmnd {
 2:    public:
 3:    public: // constructors...
 4:      TimeTone(bool sourceIsGps=TRUE);
 5:      TimeTone& TimeTone(const &TimeTone);
 6:    public: // destructor...
 7:      virtual ~TimeTone();
 8:    public:
 9:      TimeTone& operator = (const TimeTone&);
10:    public:
11:      bool     sourceIsGps() const;
12:      unsigned timeIs()      const;
13:    };
```

4.12.1 Constructor synopsis

`TimeTone` This constructor builds a so-called *time-tone* ancillary GLAST telecommand as specified in [8]. The *first* argument specifies both the execution time of the telecommand and the and the exact time corresponding to the *next* transmission of the 1-PPS signal sent to the LAT. As these two values are the same, there is no specific access to this field, however, its value can be retrieved by calling the execution time function associated with the telecommand (see Section 4.4). The *second* argument is a boolean specifying whether or not the time indicated by the command was synchronized by the VSC's GPS system. If not synchronized, time was derived by a local clock whose stability with respect to the 1-PPS signal is not guaranteed. The default option specifies that the time *was* derived through GPS. Ancillary telecommands are transmitted for execution by objects of the class described in Section 7.2. This constructor throws *no* exceptions.

4.12.2 Member synopsis

`sourceIsGps` Returns a boolean specifying whether the time indicated by the command was synchronized by the VSC's GPS receiver and can, therefore, be assumed to be stable with respect to the LAT's clock. If the value returned is `TRUE`, the telecommand was synthesized with respect to the VSC's GPS receiver.If the value returned is `FALSE`, the telecommand was synthesized with respect to the VSC's local clock. This function has *no* arguments and throws *no* exceptions.

`timeIs` Returns a value which specifies the time corresponding to the *next* transmission of the 1-PPS signal sent to the LAT. The returned value will also be equal to the time at which the telcommand will be executed, *plus* one (1). Time is represented as the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. This function has *no* arguments and throws *no* exceptions.

4.13 Exceptions

None.

Chapter 5
The Handling package

The classes of this package allow the user to register an application specific object to catch and handle incoming CCSDS packets. Objects which catch and process CCSDS packets are called *Handlers*. Incoming packets are carried on the streams discussed in Section 1.1. These packets take three different forms:

i. 1553 (Diagnostics and housekeeping) telemetry
ii. Science telemetry which includes both science housekeeping and events
iii. Telecommands

For each of type of packet there is a corresponding handler, however, all three handlers are derived through a common base class (see Section 5.2), as they all have exactly the same form except for the type of the packet they are intended to handle. The user is expected to sub-class from one of these three handlers in order to construct their own specific handler. All handlers execute in the context of a specific *thread* (see Section 3.5.1). The dependencies of the classes of this package are described in Figure 24:

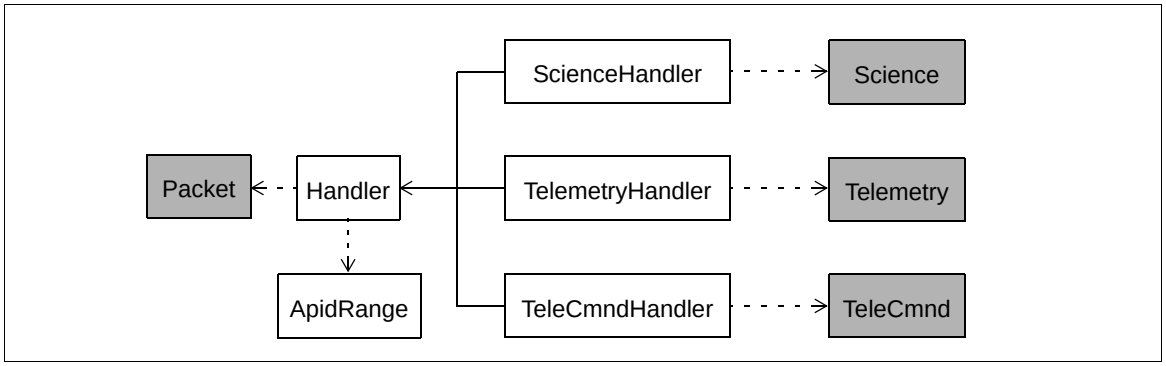

Figure 24 Class dependencies for the Handler package

5.1 Name space - `VscHandling`

5.2 Handler

This class, which forms a base class for the specific handlers described below, is used to catch and process incoming CCSDS packets. To *catch* a packet requires the derived class to specify a set of APIDs which the handler is willing to catch. The handler maintains a list of `APID` ranges (see Section 5.6) in order to determine which packets the handler will, or will not handle. To *process* a packet requires the derived class to provide an implementation of the abstract method `process`. This method is passed, as an argument, the packet to consume.

Packets arrive asynchronously on three different telemetry streams (see Section 7.2). An application makes the connection between their handler and packets arriving on a stream by registering their handler with an application specific router (see Section 6.2). In turn, this router is registered with a stream.

The class specification for the abstract handler is found in Listing 12. An application is not expected to directly inherit from this class. Instead, an application will sub-class from one of these three specific handlers:

— `TelemetryHandler` (see Section 5.3)

— `ScienceHandler`(see Section 5.4)

— `TeleCmndHandler` (see Section 5.5)

Listing 12 Class definition for `Handler`

```
 1: template<class T> Handler : public VscList::Link<Handler> {
 2:   protected: // constructors...
 3:     Handler();
 4:   public:
 5:     virtual ~Handler();
 6:   public:
 7:     virtual void process(T&) = 0;
 8:   public:
 9:     Handler<T&> next() const;
10:   public:
11:     bool catchable(const T&) const;
12:   public:
13:     const ApidRange* head() const;
14:     const ApidRange* last() const;
15:     void             handle(unsigned short);
16:     void             handle(unsigned short low, unsigned short high);
17: };
```

5.2.1 Constructor synopsis

`Handler` The constructor sets its internal list of `APID` ranges (see Section 5.6) to *empty*. This list corresponds to the set of `APID`s whose corresponding packets the handler is willing to catch. If the list is empty, the handler will catch *any* packet, independent of `APID`. This constructor throws no exceptions.

5.2.2 Member synopsis

`process` This method would be called whenever the `catchable` function (see below) matches a CCSDS packet against its `APID` list. The argument to the method specifies a reference to the packet to be processed. This function returns no value and throws no exception.

`next` If this handler resides on a router list (see Section 6.2), this method returns a reference to the handler on the list *following* this handler. If the handler is not on a list, the method returns a reference to itself. If the handler is at a router's tail, it returns a reference which is equal to the list itself. This function throws no exception.

`catchable` This method determines whether the handler could catch and process a particular incoming CCSDS packet. This packet is represented by the input argument, which is a reference to a CCSDS packet (see Section 4.2). This involves traversing the handler's internal range list. For each range on this list, if the packet's `APID` falls within that range, the handler should catch the packet. If the range list is *empty*, the handler would catch any packet. If the packet would be caught, this method returns a `TRUE` value. If the packet would not be caught, the method returns a `FALSE` value. This function throws no exceptions.

`head` This method returns a reference to the `APID` range at the head of its range list. If the list is *empty*, the reference returned is equal to the reference returned by the `last` member (see below). This function throws no exceptions.

`last` This method returns a reference to the range list. By comparing, for equality, this reference to ranges on the handler's list (see the `head` method), the user can determine whether a range is at the tail of the list. This function throws no exceptions.

`handle(unsigned short)` This method allocates and constructs an `ApidRange` (see Section 5.6) using a constructor appropriate for creating a range corresponding to the single argument. The range is inserted at the tail of the list of the handler's `APID` range list. This method returns no value and throws no exceptions.

`handle(unsigned short low, unsigned short high)` This method allocates and constructs an `ApidRange` (see Section 5.6) using a constructor appropriate for creating a range corresponding to the two arguments. The created range is inserted at the tail of the handler's `APID` range list. The function returns a reference to the created range. This method returns no value and throws no exceptions.

5.3 Telemetry Handler

This class specifies a handler for a *telemetry* packets (see Section 4.7). The implementation of this class simply derives from `Handler` (see Section 5.2) and satisfies the template argument with the class representing a telemetry packet. Any application which handles telemetry packets is expected to sub-class from `TelemetryHandler` and provide an implementation of its `process` method.

Listing 13 Class definition for `TelemetryHandler`

```
1:  class TelemetryHandler : public Handler<VscCcsds::Telemetry> { };
```

5.3.1 Constructor synopsis

see Section 5.2.

5.3.2 Member synopsis

see Section 5.2.

5.4 Science Handler

This class specifies a handler for a *science* packets (see Section 4.8). The implementation of this class simply derives from `Handler` (see Section 5.2) and satisfies the template argument with the class representing a science packet. Any application which handles science packets is expected to sub-class from `ScienceHandler` and provide an implementation of its `process` method.

Listing 14 Class definition for `ScienceHandler`

```
1:  class ScienceHandler : public Handler<VscCcsds::Science> { };
```

5.4.1 Constructor synopsis

see Section 5.2.

5.4.2 Member synopsis

see Section 5.2.

5.5 Telecommand Handler

This class specifies a handler for a *telecommand* packets (see Section 4.4). The implementation of this class simply derives from `Handler` (see Section 5.2) and satisfies the template argument with the class representing a telecommand packet. Any application which handles telecommand packets is expected to sub-class from `TeleCmndHandler` and provide an implementation of its `process` method.

Listing 15 Class definition for `TeleCmndHandler`

```
1:  class TeleCmndHandler : public Handler<VscCcsds::TeleCmnd> { };
```

5.5.1 Constructor synopsis

see Section 5.2.

5.5.2 Member synopsis

see Section 5.2.

5.6 APID Range

This class captures the concept of an `APID` *range*. The range is represented by two numbers, one value specifying the lowest (inclusive) value of the range and the other specifying the (inclusive) highest value of the range. If a range consists of a single number, then both the low and high values of the range are equal. If an incoming packet has an `APID` which falls within a range, the packet is said to be *catchable* for that range. A handler (see Section 5.2) holds a *list* of `APID` ranges. If any one of the ranges held by a handler would catch an incoming packet, then the packet is said to be catchable by the handler.

Listing 16 Class definition for `ApidRange`

```
 1:  class ApidRange : public VscList::Link<ApidRange> {
 2:    public: // constructors...
 3:      ApidRange(unsigned short);
 4:      ApidRange(unsigned short low, unsigned short high);
 5:    public:
 6:     ~ApidRange();
 7:    public:
 8:      const ApidRange& next() const;
 9:    public:
10:      unsigned short low()  const;
11:      unsigned short high() const;
12:    };
```

5.6.1 Constructor synopsis

`ApidRange(unsigned short low)` The constructor assigns the argument as the `APID` corresponding to *both* the low and high value of the range. By convention this implies the range spans a single `APID`. The constructor throws no exceptions.

`ApidRange(unsigned short low, unsigned short high)` The constructor assigns the smaller of the two arguments as the `APID` corresponding to the low value of the range. The larger of the two arguments is assigned as the `APID` corresponding to the high range. For good style, the first argument should correspond to the smaller value and the second the larger value. The constructor throws no exceptions.

5.6.2 Member synopsis

`next` If this range resides on a handler's range list (see Section 5.2), this method returns a reference to the range on the list *following* this range. If the range is not on a list, the method returns a reference to itself. If the range is at the list's tail, it returns a reference which is equal to the list itself. This function throws no exceptions.

`low` This function returns the *smaller* of the two `APID`s specifying the range. If the range's extent corresponds to only one `APID`, the value returned will be equal to the returned value of the ***high*** function described below. This function throws no exceptions.

`high` This function returns the *larger* of the two `APID`s specifying the range. If the range's extent corresponds to only one `APID`, the value returned will be equal to the returned value of the ***low*** function described above. This function throws no exceptions.

5.7 Exceptions

none.

Chapter 6
The Routing package

The classes of this package allow the user to register an application specific object to *route* incoming CCSDS packets on a stream, to be caught by an appropriate handler. Objects which catch and route CCSDS packets are called *Routers*. Incoming packets are carried on the streams discussed in Section 1.1. These packets take three different forms:

i. 1553 (Diagnostics and housekeeping) telemetry
ii. Science telemetry which includes both science housekeeping and events
iii. Telecommands

For each of type of packet there is a corresponding router, however, all three routers are derived through a common base class (see Section 6.2), as they all have exactly the same form except for the type of the packet they are intended to route. The user is expected to sub-class from one of these three routers in order to construct their own specific router. All routers (and their corresponding handlers (see chapter 5) execute in the context of a specific *thread* (see Section 1.1). The dependencies of the classes of this package are described in Figure 25:

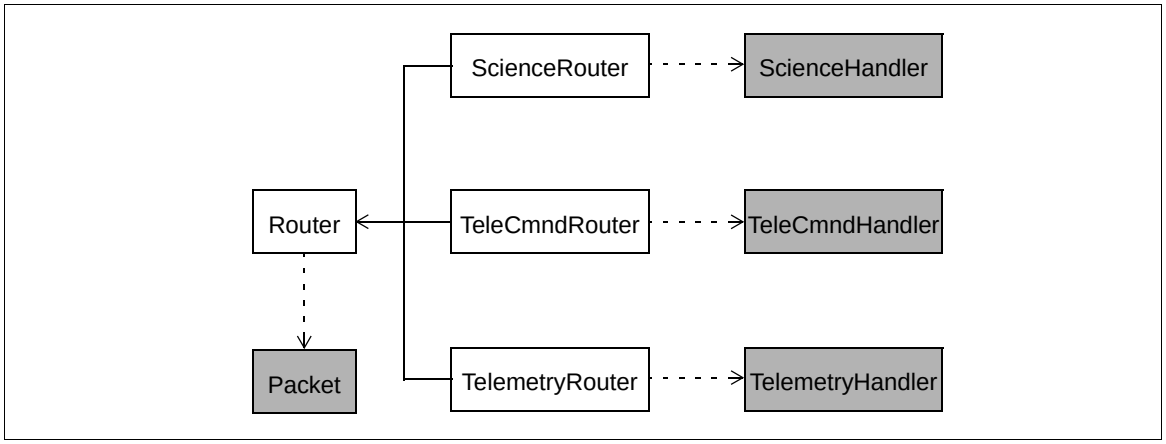

Figure 25 Class dependencies for the Routing package

6.1 Name space - `VscRouting`

6.2 Router

This class, which forms an base class for the specific routers described below, is used to route incoming CCSDS packets carried on a stream (see for example, the `Proxy` class described in Section 7.2). A stream delegates two responsibilities to a router in order to handle packets on their arrival. These are:

i. Packet memory allocation. The stream's input port calls back an application derived function (`allocate`) in order to place the arrived packet into a user specified area. In this fashion, the stream relinquishes memory allocation responsibility and passes that responsibility to the router. This allows the router to pick a memory management strategy appropriate to application. For example, if a packet could be processed and dismissed entirely by the router, storage for a single packet is all that is necessary and could be allocated within the router itself. If on the other hand, arrived packets are meant to be passed through the router to other objects for processing, then allocation from a free-store would be more appropriate.

ii. Packet catching. Once a packet has arrived and been placed in the memory specified by the router, the stream's input port calls back the router's `route` method in order to dispose of the arrived packet. This method searches for a handler (see Section 5.2) to catch and process the arrived packet.

The class specification for the abstract router is found in Listing 17. An application is not expected to directly inherit from this class. Instead, an application will sub-class from one of three type specific routers:

— `TelemetryRouter` (see Section 6.3)

— `ScienceRouter` (see Section 6.4)

— `TeleCmndRouter` (see Section 6.5)

Listing 17 Class definition for `Router`

```
 1: template<class T> class Router {
 2:   public: // constructors...
 3:     Router();
 4:   public:
 5:     virtual ~Router();
 6:   public:
 7:     virtual T*   allocate()   = 0;
 8:     virtual void catchall(T&) = 0;
 9:   public:
10:     void route(T&);
11:   public:
12:     const VscHandling::Handler<T>* head() const;
13:     const VscHandling::Handler<T>* last() const;
14:   public:
15:     void insert(VscHandling::Handler<T>&);
16:   };
```

6.2.1 Constructor synopsis

`Router` The constructor sets its internal list of handlers (see Section 5.2) to *empty*. This list corresponds to the set of handlers, which the router is prepared to dispatch packets to. If the list is empty, the router will catch *any* packet, independent of `APID`, by calling back its `catchall` method. This constructor throws no exceptions.

6.2.2 Member synopsis

`allocate` This function is used to locate the memory to place an incoming CCSDS packet. This function is always called *before* the appropriate handler is fired. If memory cannot be allocated, the derived class must throw the exception described in 6.6.1. The function returns a *pointer* to the allocated packet.

`route` This method attempts to catch and process a particular incoming CCSDS packet. This packet is represented by the input argument, which is a reference to a CCSDS packet (see Section 4.3). This involves traversing the routers's internal handler list, checking for a handler to catch the packet. If a handler is not found on the list, the `catchall` method (see below) is invoked, passing the caught packet as an argument. This function returns no value and throws no exceptions.

`catchall` This function is called by the router's `route` method (see above) whenever either its handler list is empty or, it cannot find a handler on its list willing to catch the CCSDS packet associated with the call to the `catch` method. The argument to the method specifies a reference to the corresponding packet. This function returns no value and throws no exception.

head This method returns a pointer to the `handler` at the head of the router's handler list. If the list is *empty,* the pointer returned is equal to the reference returned by the `last` member (see below).This function throws no exception.

last This method returns a reference to the router's handler list. By comparing, for equality, this return value to the handlers on the list (see the `head` method), the user can determine whether the handler is the range at the tail of the list. This function throws no exception.

insert This method inserts the handler specified by its argument at the *tail* of the handler's own list. The argument is a reference to the handler (see Section 5.2) to insert on the list. The method returns a reference to the inserted handler. This function throws no exceptions.

6.3 Telemetry Router

This class specifies a router for a *telemetry* packets arriving on a telemetry *stream* (see Section 7.2). The implementation of this class simply derives from `Router` (see Section 6.2) and satisfies the template argument with the class representing a telemetry packet (see Section 4.3). Any application which routes telemetry packets is expected to sub-class from `TelemetryRouter` and provide an implementation of both its `allocate` and `catchall` methods.

Listing 18 Class definition for `TelemetryRouter`

```
1:  class TelemetryRouter : public Router<VscCcsds::Telemetry> {};
```

6.3.1 Constructor synopsis

see Section 6.2

6.3.2 Member synopsis

see Section 6.2

6.4 Science Router

This class specifies a router for a *science* packets arriving on a science *stream* (see Section 7.2). The implementation of this class simply derives from `Router` (see Section 6.2) and satisfies the template argument with the class representing a science packet (see Section 4.8). Any

application which routes science packets is expected to sub-class from `ScienceRouter` and provide an implementation of both its `allocate` and `catchall` methods.

Listing 19 Class definition for `ScienceRouter`

```
1:  class ScienceRouter : public Router<VscCcsds::Science> {};
```

6.4.1 Constructor synopsis

see Section 6.2

6.4.2 Member synopsis

see Section 6.2

6.5 Telecommand Router

This class specifies a router for a *telecommand* packets arriving on a telecommand *stream* (see Section 7.2). The implementation of this class simply derives from `Router` (see Section 6.2) and satisfies the template argument with the class representing a telecommand packet (see Section 4.4). Any application which routes telecommand packets is expected to sub-class from `TeleCmndRouter` and provide an implementation of both its `allocate` and `catchall` methods.

Listing 20 Class definition for `TeleCmndRouter`

```
1:  class TeleCmndRouter : public Router<VscCcsds::TeleCmnd> {};
```

6.5.1 Constructor synopsis

see Section 6.2

6.5.2 Member synopsis

see Section 6.2

6.6 Exceptions

6.6.1 Insufficient Memory

To be written.

Chapter 7
The VSC proxy package

The VSC package consists of the classes which will form the anchor of any user application. The classes contained within this package have four major functions:

i. Provide a remote, network passed connection to the VSC. This is accomplished by instantiating the `Proxy` class.

ii. Receive telemetry packets originating from both the LAT (through the VSC) and the VSC itself. This is accomplished by registering telemetry routers (see Chapter 6), using member functions of the `Proxy` class.

iii. Command the VSC. This is accomplished by calling member functions of the `Proxy` class passing, as arguments, specific request telecommands to be queued to and executed by the VSC. *All* request telecommands inherit from the `Request` class. The bulk of the classes described in this chapter are request telecommands.

iv. Command the LAT. This is accomplished by calling member functions of the `Proxy` class passing, as arguments, LAT specific telecommands to be queued to the VSC for subsequent transmission over its 1553 interface to the LAT.

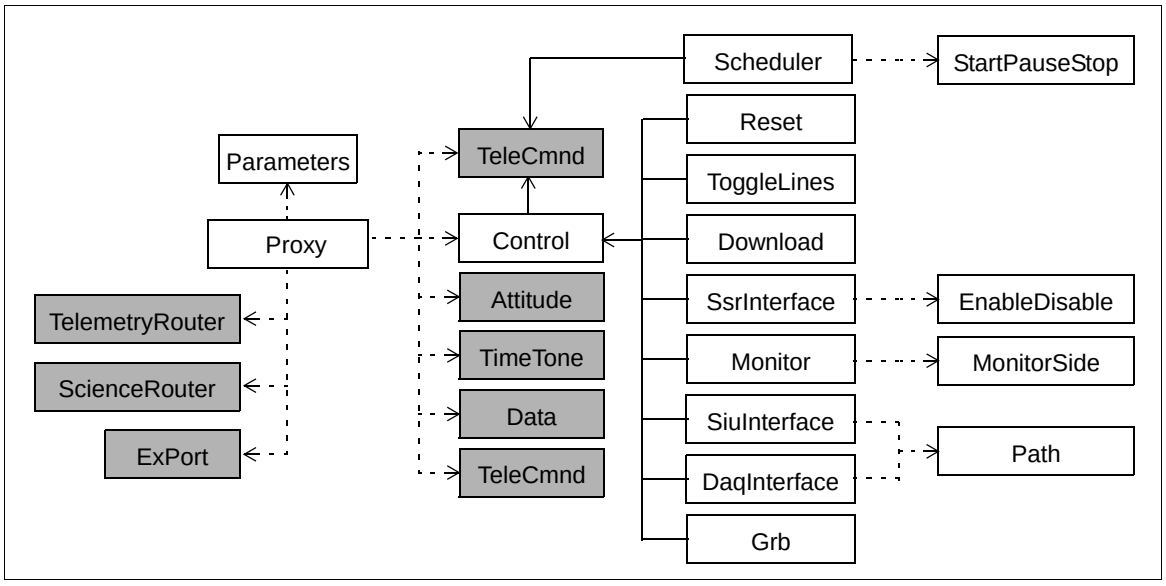

Figure 26 Class dependencies for the proxy package

7.1 Name space - `VscProxy`

7.2 Proxy

The `Proxy` class provides a remote, network based interface to the functionality of the VSC. The network protocol is packet oriented and layered on standard TCP/IP connections. The act of instantiating the `Proxy` class creates all appropriate network connections to the VSC. Because of their pervasive usage throughout the observatory the "on wire" packet protocol is based on CCSDS (see [17] and [18]) packets. The proxy can be thought of as the nexus of six different network streams. Five of these streams transmit down-linked packets (from the VSC to the proxy) and one transmits up-linked information (from the proxy to the VSC). Up-linked information consists of CCSDS telecommands. These telecommands fit into two generic classes:

Requests: Directions to the VSC itself. Each individual request has a corresponding class, however, all vsc requests inherit from the same base class (`Request`).

TeleCmnds: Directions to the LAT, which are sent through the VSC. The base class for LAT specific telecommands is defined in Section 4.4.

VSC requests and LAT telecommands are transmitted by the application to the VSC by calling the `schedule` and `scheduleAt` member functions. The arguments to these functions consist of either requests or LAT telecommands to be sent to the VSC. Both functions allow an application to package up and transmit *sets* of packets. Once arrived on the VSC these packets may be either queued for immediate action or transmission by the VSC (implicitly identified

by using the `schedule` function), or may be stored for subsequent action or transmission at a specific time (implicitly identified by using the `scheduleAt` function).

Down-linked information is encoded as GLAST telemetry packets (see Chapter 4). The application receives this telemetry by registering a router see Chapter 6) for each of the five, asynchronous telemetry streams.

Listing 21 Class definition for `Proxy`

```
 1: class Proxy {
 2:   public:
 3:     Proxy();
 4:     Proxy(Parameters&);
 5:   public:
 6:    ~Proxy();
 7:   public:
 8:     void vscDiagnostic(VscRouting::TelemetryRouter&);
 9:     void vscTelemetry(VscRouting::TelemetryRouter&);
10:     void latDiagnostic(VscRouting::TelemetryRouter&);
11:     void latTelemetry(VscRouting::TelemetryRouter&);
12:     void latScience(VscRouting::ScienceRouter&);
13:   public:
14:     void execute(const Scheduler&);
15:     void execute(const Control&...);
16:     void execute(const VscCcsds::TeleCmnd&...);
17:   public:
18:     void executeAt(unsigned time, const Control&...);
19:     void executeAt(unsigned time, const VscCcsds::TeleCmnd&...);
20:   public:
21:     void schedule();
22:     void schedule(const VscCcsds::Attitude&,
23:                   const VscCcsds::TimeTone&,
24:                   const VscCcsds::Attitude&,
25:                   const VscCcsds::Attitude&,
26:                   const VscCcsds::Attitude&,
27:                   const VscCcsds::Data&,
28:                   const VscCcsds::Attitude&);
29:     void schedule(unsigned time, const VscCcsds::Attitude&,
30:                                  const VscCcsds::TimeTone&,
31:                                  const VscCcsds::Attitude&,
32:                                  const VscCcsds::Attitude&,
33:                                  const VscCcsds::Attitude&,
34:                                  const VscCcsds::Data&,
35:                                  const VscCcsds::Attitude&);
36:   };
```

7.2.1 Constructor synopsis

`Proxy()` This constructor creates an output port which is then connected to the VSC's *command* stream. In order to successfully connect this stream requires the VSC's node name and command port number. These parameters are specified by an instance of the class described in Section 7.14. The constructor has no arguments. The constructor throws the exceptions: Section 7.15.2 and Section 7.15.3.

`Proxy(Parameters&)` This constructor creates two output ports which are then connected to the VSC's *control* stream and *command* streams. In order to successfully connect these two streams requires the VSC's node name, control and command port numbers. These parameters are specified by an instance of class described in Section 7.14. The application can over-ride one or more these parameters by providing its own instance of the proxy's parameters. The argument is a reference to an object containing these parameters. The constructor will replace its own internal copy of these parameters with the parameters specified by the argument. The constructor throws the exceptions: Section 7.15.2 and Section 7.15.3.

7.2.2 Member synopsis

`vscDiagnostic` Creates a connection to the VSC's *diagnostic* stream and registers an application specific router to process the diagnostic telemetry acquired by the VSC and returned through the stream. The set of potential diagnostic telemetry returned by the VSC is described in appendix B. The router's code is executed in the context of the thread named "`VscDiagnosticThread`". The priority of the thread is specified by adding *zero* (0) to a common base priority. The base priority is returned by the `basePriority` function of the class described in Section 7.14. The functions's argument specifies a reference to the router to process incoming packets (see Section 6.2). The router will be called back once for every received CCSDS telemetry packet. One can only register a router *once* per stream. If this member function has been previously called, the exception described in Section 7.15.1 will be thrown. If this member function cannot transmit the connection request to the VSC due to a network error, the exception described in Section 7.15.3 will be thrown. This member function returns no value.

`vscTelemetry` Creates a connection to the VSC's *telemetry* stream and registers an application specific router to process the telemetry acquired by the VSC and returned through the stream. The set of potential telemetry returned by the VSC is described in xxx. The router's code is executed in the context of the thread named "`VscTelemetryThread`". The priority of the thread is specified by adding *two* (2) to a common base priority. The base priority is returned by the `basePriority` function of the class described in Section 7.14. The functions's argument specifies a reference to the router to process incoming packets (see Section 6.2). One can only register a router *once* per stream. If this member function has been previously called, the exception described in Section 7.15.1 will be thrown. If this member function cannot transmit the connection request to the VSC due to a network error, the exception described in Section 7.15.3 will be thrown. This member function returns no value.

`latDiagnostic` Creates a connection to the VSC's *LAT diagnostic* stream and registers an application specific router to process the diagnostic telemetry acquired by the LAT and returned through the stream. The router's code is executed in the context of the thread named "`VscLatDiagnosticThread`". The priority of the thread is specified by adding *one* (1) to a common base priority. The base priority is returned by the `basePriority` function of the class described in Section 7.14. The functions's argument specifies a pointer to the router to process incoming packets (see Section 6.2). The router will be called back once for every received CCSDS telemetry packet. One can only register a router *once* per stream. If this member function has been previously called, the exception described in Section 7.15.1 will be thrown. If this member function cannot transmit the connection request to the VSC due to a network error, the exception described in Section 7.15.3 will be thrown. This member function returns no value.

`latTelemetry` Creates a connection to the VSC's *LAT telemetry* stream and registers an application specific router to process the telemetry acquired by the LAT and returned through the stream This includes both the so-called LAT housekeeping and diagnostic telemetry. The router's code is executed in the context of a thread named "`VscLatTelemetryThread`". The priority of the thread is specified by adding *three* (3) to a common base priority. The base priority is returned by the `basePriority` function of the class described in Section 7.14. The functions's argument specifies a reference to the router to process incoming packets (see Section 6.2). The router will be called back once for every received CCSDS telemetry packet. One can only register a router *once* per stream. If this member function has been previously called, the exception described in Section 7.15.1 will be thrown. If this member function cannot transmit the connection request to the VSC due to a network error, the exception described in Section 7.15.3 will be thrown. This member function returns no value.

`latScience` Registers an application specific router to process telemetry received by the VSC on the *science* telemetry stream. The router's code is executed in the context of a thread named "`VscScienceThread`". The priority of the thread is specified by adding *four* (4) to a common base priority. The base priority is returned by the `basePriority` function of the class described in Section 7.14. The functions's argument specifies a reference to the router to process incoming packets (see Section 6.2). The router will be called back once for every received CCSDS telemetry packet. One can only register a router *once* per stream. If this member function has been previously called, the exception described in Section 7.15.1 will be thrown. If this member function cannot transmit the connection request to the VSC due to a network error, the exception described in Section 7.15.3 will be thrown. This member function returns no value.

`execute(Scheduler&)` Transmits a scheduler requests on the command stream to the VSC, where these requests are then queued on the VSC for subsequent execution as soon as the VSC has the capability to do so. The scheduler requests to execute are represented by the function's arguments. Relative to each other, these commands are executed by the VSC in argument order. The relative time structure between the execution of these requests and other telecommands transmitted to the LAT is

discussed in Section 1.4. The arguments are references to a series of objects. Each object corresponds to a scheduler request (as a telecommand, see Section 7.3) to be sent to the LAT. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`execute(Control&,...)` Transmits a set of up to *seven* (7) command requests on the command stream to the VSC, where these requests are then queued on the VSC for subsequent execution as soon as the VSC has the capability to do so. The command requests to execute are represented by the function's arguments. Relative to each other, these commands are executed by the VSC in argument order. The relative time structure between the execution of these requests and other telecommands transmitted to the LAT is discussed in Section 1.4. The arguments are references to a series of objects. Each object corresponds to a command request (as a telecommand, see Section 7.4) to be sent to the LAT. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`execute(TeleCmnd&,...)` Transmits a set of up to *five* (5) commands on the command stream to the VSC, where these telecommands are then queued on the VSC for subsequent transmission to the LAT as soon as the VSC has the capability to do so. The commands necessary to transmit are represented by the function's arguments. Relative to each other, these commands are transmitted to the LAT in argument order. The relative time structure between these and other telecommands transmitted to the LAT is discussed in Section 1.4. The arguments are references to a series of objects. Each object corresponds to a telecommand (see Section 4.4) to be sent to the LAT. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`executeAt(unsigned time, Control&,...)` Transmits a set of up to *seven* (7) command requests on the command stream to the VSC, where these requests are then queued on the VSC for subsequent execution at a time within a specific one second period. The period is specified by the first argument, whose value represents the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The command requests to execute are represented by the arguments following the period (first) argument. Relative to each other, these requests are transmitted to the LAT in argument order. The relative time structure between the execution of these requests and other telecommands transmitted to the LAT is discussed in Section 1.4. The arguments following the first argument are references to a series of objects. Each object corresponds to a command request (sent as a telecommand, see Section 4.4) to be executed by the VSC within the scheduled one second period. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`executeAt(unsigned time, TeleCmnd&,...)` Transmits a set of up to *eight* (8) commands on the command stream to the VSC, where these telecommands are then queued on the VSC for subsequent transmission to the LAT at a time within a specific one second period. The period is specified by the first argument, whose value represents the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The commands transmitted are represented by the arguments following the period (first) argument. Relative to each other, these commands are transmitted to

the LAT in argument order. The relative time structure between these and other telecommands transmitted to the LAT is discussed in Section 1.4. The arguments following the first argument are references to a series of objects. Each object corresponds to a telecommand (see Section 4.4) to be sent to the LAT within the scheduled one second period. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`schedule()` Disables sending the default ancillary sequence. See the member function below. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`schedule(Attitude&,...)` Transmits a set of seven *specific* commands on the command stream to the VSC, where these telecommands form the default set of ancillary commands. The first argument is a reference to the first of the five different, default attitude packets (see Section 4.10). The second argument is a reference to the default time-tone packet (see Section 4.12). The third, fourth, and fifth arguments are references to the next three default attitude packets The sixth argument is a reference to the single, default ancillary data packet (see Section 4.11). The seventh (and last) argument is a reference to the fifth (and last) default attitude packet. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

`schedule(unsigned time, Attitude&,...)` Transmits a set of seven *specific* commands on the command stream to the VSC, where these telecommands are then queued on the VSC for subsequent transmission to the LAT at a time within a specific one second period. The period is specified by the first argument, whose value represents the number of seconds since the epoch 00:00:00.0 hours at January 1st, 2001. The seven commands transmitted are represented by the seven arguments following the period argument. Relative to each other these commands are transmitted to the LAT in argument order. The relative time structure between these and other telecommands transmitted to the LAT is discussed in Section 1.4. The second argument is a reference to the first of the five different attitude packets (see Section 4.10) sent to the LAT within the scheduled one second period. The third argument is a reference to the time-tone packet sent to the LAT period (see Section 4.12)within the scheduled one second period. The fourth, fifth, and sixth arguments are references to the next three attitude packets sent to the LAT within the scheduled one second period. The seventh argument is a reference to the single ancillary data packet (see Section 4.11) sent to the LAT within the scheduled one second period. The eight (and last) argument is a reference to the fifth (and last) attitude packet sent to the LAT within the scheduled one second period. This function returns no value. This function can throw either one of the exceptions described in Section 7.15.2. and Section 7.15.3.

7.3 Scheduler control request

Constructs a change scheduler state request. The new state is specified as an enumeration passed as an argument to the constructor.

Note: If this request specifies resetting the time-base can only be issued if the VSC is currently in a stopped state (see Section 1.4.4)

Listing 22 Class definition for `Scheduler`

```
 1:  class Scheduler : VscCcsds::TelCmnd {
 2:    public:
 3:      enum StartPauseStop {Start = 1, Pause = 2, Stop = 3};
 4:    public:
 5:      Scheduler(Scheduler::StartPauseStop, unsigned timebase = 0);
 6:   public:
 7:     ~Scheduler();
 8:    public:
 9:      Scheduler::StartPauseStop state() const;
10:    public:
11:      unsigned time() const;
12:    };
```

7.3.1 Constructor synopsis

`Scheduler` This constructor takes as its first argument an enumeration expressing the state which the scheduler is to be driven. See Section 1.4.4 for the meaning of each scheduler state. The second argument specifies the new, initial value of the time-base, as measured as the time since the standard epoch (00:00:00.0 hours at January 1st, 2001). Time is measured in units of seconds. A value of *zero* (the default), specifies the time-base should be set to the current value of the wall-clock time-base (see Section 1.3.1).The constructor throws no exception.

7.3.2 Member synopsis

`state` Returns the requested state, expressed as an enumeration. The member function has no arguments and throws no exceptions. If the request was to reset the time-base, the function will return the enumeration `Stop`.

`time` Returns the initial value of the time-base on the VSC, as the time since the standard epoch (00:00:00.0 hours at January 1st, 2001) in seconds. This function has no arguments and throws no exceptions. If the request did not involve resetting the time-base, this function will return a value of *zero* (0).

7.4 Control request

All telecommands which provide direction to the VSC are called *control* requests. In order to differentiate request based telecommands from GLAST based telecommands, specific requests (described below) all sub-class from `Request`. Usage of this class is reserved to the implementation. An application should never directly either use or sub-class from `Control`. It is documented in this chapter only for completeness.

Listing 23 Class definition for `Control`

```
1:  class Control : VscCcsds::TelCmnd {
2:      Request(unsigned short apid, unsigned short function);
3:    public:
4:     ~Request();
5:    };
```

7.5 Cross-strapping options

In order to mitigate against single point-failure both the VSC and LAT have redundant SIU and and DAQ communication interfaces. All these interfaces are cross-strapped and any change to the current cross-strapping is governed by the request classes described in sections 7.6 and 7.7. Each of two classes takes as an argument, the enumeration defined below:

```
enum Path {AA = 1, AB = 2, BA = 3, BB = 4};
```

This enumeration specifies which one of the four cross strapping options should be applied as enumerated in Table 9:

Table 9 Cross-strapping options

Path	Use Unit?			
	VSC		LAT	
	Primary	Redundant	Primary	Redundant
AA	yes	no	yes	no
AB	yes	no	no	yes
BA	no	yes	yes	no
BB	no	yes	no	yes

7.6 SIU interface control request

The LAT has two SIUs (Spacecraft Interface Unit). One SIU is designated as the *Primary* SIU and the other as the *Redundant* SIU. In orbit, the redundant SIU is called out as a cold spare. As the name implies, SIUs interact with the spacecraft. In turn, the spacecraft mitigates against single-point failure by having *two* SIU interfaces which the VSC emulates. One interface is called out as the *Primary* LAT interface and the other as the *Redundant* LAT interface. In order to support full redundancy each of the four units has both an "A" and "B" port. The SIUs and VSC interfaces are physically cross-strapped in order to allow mixing and matching of the four units. This request is used to specify which one of the four different cross-strapping options should be used. See Section 3.3.1 for more information.

Listing 24 Class definition for `SiuInterface`

```
1:  class SiuInterface : public Control {
2:     public:
3:       SiuInterface(VscProxy::Path = AA);
4:    public:
5:     ~SiuInterface();
6:    public:
7:       VscProxy::Path path() const;
8:    };
```

7.6.1 Constructor synopsis

`SiuInterface` This constructor takes as an argument the enumeration described in Section 7.5. This enumeration determines the exact cross-strapping option requested. If the argument is omitted the cross-strapping request is equal to the initial cross-strapping established by the VSC. The constructor throws no exceptions.

7.6.2 Member synopsis

`path` Returns the enumeration described in Section 7.5. This enumeration specifies which one of the four cross-strapping options is implied by the request. This function has no arguments and throws no exceptions.

7.7 DAQ interface control request

The LAT has one GASU which contains two DAQ boards. One DAQ board is designated as the *Primary* DAQ board and the other as the *Redundant* DAQ board. Only one of the two DAQ boards is active at any one time. In turn, the spacecraft mitigates against single-point failure by having *two* DAQ board interfaces. One interface is called out as the *Primary* LAT interface

and the other as the *Redundant* LAT interface which the VSC emulates. In order to support full redundancy each of the four units has both an "A" and "B" port. The DAQ boards and VSC interfaces are physically cross-strapped in order to allow mixing and matching of the four units. This request is used to specify which one of the four different cross-strapping options should be used. See Section 3.3.1 for more information.

Listing 25 Class definition for `DaqInterface`

```
1:  class DaqInterface : public Control {
2:     public:
3:      DaqInterface(VscProxy::Path = AA);
4:    public:
5:     ~DaqInterface();
6:    public:
7:      VscProxy::Path path() const;
8:    };
```

7.7.1 Constructor synopsis

`DaqInterface` This constructor takes as an argument the enumeration described in Section 7.5. This enumeration determines the exact cross-strapping option requested. If the argument is omitted the cross-strapping request is equal to the initial cross-strapping established by the VSC. The constructor throws no exceptions.

7.7.2 Member synopsis

`path` Returns the enumeration described in Section 7.5. This enumeration specifies which one of the four cross-strapping options is implied by the request. This member has no arguments and throws no exceptions.

7.8 SIU discrete control request

The function of this request is change the state of one or more of the SIU's discrete lines.

Note: This interface allows a discrete line to be either asserted or deasserted. The definition of assertion follows the LAT negative logic convention, where a TRUE (asserted) logical value corresponds to a

physical FALSE (0) value and a FALSE (deasserted) logical value corresponds to a physical TRUE (1) value.

Listing 26 Class definition for `ToggleLines`

```
 1: class ToggleLines : public Control {
 2:   public:
 3:     enum Lines {Line0=0x1, Line1=0x2, Line2=0x4};
 4:   public:
 5:     ToggleLines(unsigned lines, unsigned values);
 6:   public:
 7:    ~ToggleLines();
 8:   public:
 9:     unsigned lines()  const;
10:     unsigned values() const;
11:   };
```

7.8.1 Constructor synopsis

`ToggleLines` The first argument corresponds to a 3-bit mask which defines which of the three discrete lines are to be changed. Each bit offset corresponds to a line. If the bit at the corresponding offset is *set*, the value of the line is to be changed. If the bit at the offset is *clear*, the line's value is left unchanged. The new values of the discrete lines are determined by the second argument. Each bit offset of this argument also corresponds to a discrete line. If the bit at the corresponding offset is *set*, and the corresponding offset in the first argument is *set*, the discrete line is *asserted*. If the bit at the corresponding offset is *clear*, and the corresponding offset in the first argument is *set*, the discrete is *deasserted*. The class provides an enumeration for each of the bit offsets.

7.8.2 Member synopsis

`lines` Returns the discrete lines to be changed. Each bit offset corresponds to a line. If the bit at the corresponding offset is *set*, the value of the line is to be changed. If the bit at the offset is *clear*, the line's value is left unchanged. The new values of the discrete lines are determined by the `values` member function described below. This function has no arguments and throws no exceptions.

`values` Returns the values of the discrete lines changed. The discrete lines changed are determined by the `lines` function described above. Each bit offset corresponds to a line. If the bit at a specified offset is *set*, and the corresponding bit offset in the returned value from the `lines` function is *set*, the line is to be *enabled*. If the bit at a specified offset is *clear*, and the corresponding bit offset in the returned value from the `lines` function is *set*, the line is to be *disabled*. This function has no arguments and throws no exceptions.

7.9 SIU reset control request

The function of this request is to issue a reset to the currently selected SIU, on its currently selected path. See Section 3.3.1 for a discussion on how the SIU and its path are selected.

Listing 27 Class definition for `Reset`

```
1:  class Reset : public Control {
2:    public:
3:      Reset();
4:    public:
5:     ~Reset();
6:    };
```

7.9.1 Constructor synopsis

default.

7.9.2 Member synopsis

none.

7.10 Monitor control request

The function of this request is to either enable or disable the monitoring of LAT analog (voltage and temperature) data (see [7]). When enabled, monitoring information is sent on the VSC diagnostic stream (see Section 7.2) as a series of CCSDS packets. The number and structure of these packets are described in Appendix B. Their are two identical monitors: The *Primary* monitor and the *Redundant* Monitor. Only one of these two monitors may be enabled at any one time. By default, when the VSC is started, monitoring is *disabled*.

Listing 28 Class definition for `Monitor`

```
 1:  class Monitor : public Control {
 2:    public:
 3:      enum MonitorSide {None, Primary, Redundant};
 4:    public:
 5:      Monitor(MonitorSide = None);
 6:    public:
 7:     ~Monitor();
 8:    public:
 9:      MonitorSide side() const;
10:    };
```

7.10.1 Constructor synopsis

`Monitor` This constructor takes as an argument an enumeration which determines whether or not monitoring is to be *enabled* or *disabled*. If enabled, the enumeration specifies whether to enable the *Primary* or *Redundant* monitor. This constructor throws no exceptions.

7.10.2 Member synopsis

`side` Returns the requested monitor, expressed as an enumeration. The member function has no arguments and throws no exceptions.

7.11 SSR control request

The function of this request is to either enable or disable the reception of data through the SSR Interface. Enabling or disabling the interface corresponds to *asserting* or *deasserting* the "`DEVICE READY`" line of the interface (see [7]). If the interface was already enabled, this operation has no effect.

Listing 29 Class definition for `SsrInterface`

```
 1:  class SsrInterface : public Control {
 2:    public:
 3:      enum EnableDisable {Enable, Disable};
 4:    public:
 5:      SsrInterface(EnableDisable = Disable);
 6:    public:
 7:     ~SsrInterface();
 8:    public:
 9:      EnableDisable state() const;
10:    };
```

7.11.1 Constructor synopsis

`SsrInterface` This constructor takes as an argument an enumeration (see Section 7.11) which determines whether or not the interface is to be *enabled* or *disabled*. This constructor throws no exceptions.

7.11.2 Member synopsis

`state` Returns the requested state, expressed as an enumeration. The member function has no arguments and throws no exceptions.

7.12 Down load control request

The function of this request is to be described.

Listing 30 Class definition for Download

```
1:  class Download : public Control {
2:    public: // constructors...
3:      Download(unsigned short apid);
4:    public:
5:     ~Download();
6:    public:
7:      unsigned short apid() const;
8:      unsigned long  tid()  const;
9:    };
```

7.12.1 Constructor synopsis

`Download` This constructor takes as an argument the APID of the telemetry packet to be emitted (on the VSC diagnostic stream). The set of allowed APIDs is specified by xxx. This constructor throws no exceptions.

7.12.2 Member synopsis

`apid` Returns the APID of the telemetry packet to be down-loaded. The function has arguments and throws no exceptions.

`tid` Returns the transaction ID for the request. The transaction ID is a unique number, generated by the constructor and reflected back in the emitted telemetry. The function has arguments and throws no exceptions.

7.13 GBM control request

The function of this request is to transmit a GRB pulse to the LAT.

Listing 31 Class definition for`Grb`

```
1:  class Grb : public Control {
2:    public:
3:      Grb();
4:    public:
5:     ~Grb();
6:    };
```

7.13.1 Constructor synopsis

default

7.13.2 Member synopsis

none

7.14 Proxy parameters

To be written.

Listing 32 Class definition for `Parameters`

```
 1:  class Parameters {
 2:    public: // constructors...
 3:      Parameters();
 4:    public:
 5:     ~Parameters();
 6:    public:
 7:      virtual const char* vsc();
 8:    public:
 9:      virtual int control();
10:    public:
11:      virtual int basePriority();
12:    };
```

7.14.1 Constructor synopsis

`Parameters` To be written.

7.14.2 Member synopsis

`vsc` Returns the TCP/IP host name of the VSC. Note: this function is *virtual*. While there should be no necessity to do so, if the host name agreement between VSC and proxy interface is broken, the default assignment may be changed by over-riding this function. This function has no arguments and throws no exceptions.

`control` Returns the port number of the TCP/IP port used to send outgoing CCSDS telecommand packets to the VSC. These telecommands are handled internally by the VSC. See the `command` function described below for the port number used to send telecommands which are relayed to the LAT. Note: this function is *virtual*. While there should be no necessity to do so, if the port number agreement between VSC and proxy interface is broken, the default assignment may be changed by over-riding this function. This function has no arguments and throws no exceptions.

`basePriority` Returns the priority base used to establish the priority of the three different threads used to process incoming CCSDS telemetry packets (see Section 7.2). The returned value may vary from *zero* (0) to 125 (decimal), with *increasing* priority corresponding to *decreasing* return value. The relative priority between these three threads is fixed and cannot be changed, however, as this function is virtual it may be over-ridden and the base priority re-assigned.

7.15 Exceptions

7.15.1 Data Stream Allocated

To be written.

7.15.2 No transmit port

To be written.

7.15.3 Network Transmit Failure

To be written.

Appendix A
The Datagram support package

By LAT convention, information passed on the science stream is always organized in units of *datagrams* (see [23]). A Datagram is a common structure used to encapsulate all science data created and transmitted by Flight Software (FSW). Much (but not all) of the complication of processing science as opposed to housekeeping telemetry is due to this encapsulation. For example, while in principal, datagram size is unbounded, this is not the case for CCSDS packets. To deal with this eventuality FSW transmits datagrams as a series of one or more science CCSDS packets with the *same* `APID`. In other words, a datagram may *span* CCSDS science packets. Sequencing information within the packet (see [16]) is used to identify which packet is which part of a datagram. The principal function of this package is therefore the *assembly* of CCSDS science packets into datagrams. An assembled datagram is represented by the `Datagram` class (see [27]). As packets for these datagrams arrive potentially out of order they must be sequenced before assembly. Further, as packet transmission is not considered reliable, the package must assure a mechanism to both catch and report both duplicate packets and incomplete datagrams. The classes used to support datagram assembly are all based on the `Assembler` class described in Section A.2.

The relationship between the classes of this package is illustrated in Figure A.1:

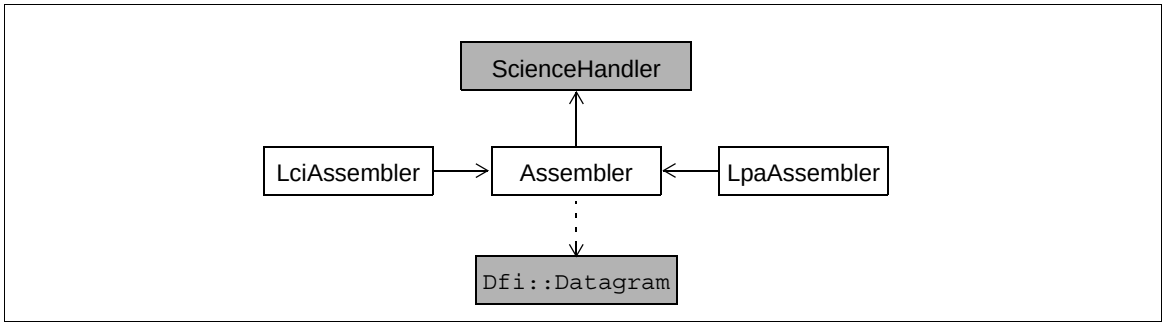

Figure A.1 Class dependencies for the datagram support package

A.1 Name space - `VscDatagram`

A.2 Datagram Assembler

This class, which forms a base class for specific handlers described below is used to catch and assemble into datagrams incoming CCSDS science packets. This class is derived from the canonical science handler (see Section 4.8). As packets arrive to assemble, the class triggers the `validate` method passing as an argument the incoming packet. Once the datagram is assembled, the `dispose` method is triggered. This method receives, as an argument, both the assembled datagram and the number of packets which were used in its assembly.

The class retains state, based on `APID`, for any each pending assembly datagram. As packets arrive they must satisfy the following constraints:

— Their sequence flag value is expected

— Their sequence number is monotonically increasing

— Their contributed size will not cause the datagram to overflow

If any of these constraints are violated the datagram cannot be successfully assembled. The class discards the runt (pending) datagram and triggers the `unexpected` method. This method receives, as an argument, the unexpected packet, the expected state[1] and sequence flags.

Typically, this class is not used directly. Instead, use one of the two derived classes: `LciAssembler` (see Section A.3) or `LpaAssembler` (see Section A.4).

The definition for this class is contained in Listing A.1:

Listing A.1 Class definition for `Assembler`

```
 1: class Assembler : public VscHandling::ScienceHandler {
 2:   public:
 3:     Assembler(unsigned maxDatagramSize);
 4:   public:
 5:     virtual ~Assembler();
 6:   public:
 7:     virtual void validate(const VscCcsds::Science&)           = 0;
 8:     virtual void dispose(const Datagram&, unsigned fragments) = 0;
 9:     virtual void unexpected(const VscCcsds::Science&,
10:                             unsigned state, unsigned seqNum)  = 0;
11:   public:
12:      unsigned maxDatagramSize() const;
13:   };
```

1. The explanation for state is TBD.

A.2.1 Constructor synopsis

`Assembler` The argument is the maximum size (in bytes) of any datagram expected to be assembled by the class. The constructor throws no exceptions.

A.2.2 Member synopsis

`validate` This method will be called whenever a CCSDS science packet arrives which is to be assembled into the resulting datagram. The default implementation is a no-op and simply returns. The argument is a reference to the arrived CCSDS science packet. Note that the argument referenced by this method (and any objects *it* may reference) is *ephemeral*. When the method returns these objects are no longer accessible. Consequently, any information in these objects which the user finds necessary to persist across calls to this method must be copied. Note: This function is pure virtual and, therefore, its implementation must be provided by a derived class.This function returns no value and throws no exceptions.

`dispose` This method is called once per assembled datagram. It will be called immediately after all the packets which constitute the datagram have been processed. The first argument is a reference to the assembled datagram (see [27]). The second argument is these number of fragments (CCSDS science packets) used in the assembly of the datagram. Note that the argument referenced by this method (and any objects *it* may reference) is *ephemeral*. When the method returns these objects are no longer accessible. Consequently, any information in these objects which the user finds necessary to persist across calls to this method must be copied. Note: This function is pure virtual and, therefore, its implementation must be provided by a derived class.This function returns no value and throws no exceptions.

`unexpected` This method will be called whenever a CCSDS science packet arrives at the assembler which was not expected. For example, the assembler may be waiting for the last packet of a sequence, and is sent instead the first packet of the sequence. The first argument is a reference to the unexpected CCSDS packet. The second argument is the expected state expressed as a value from *zero* to *fifteen* (decimal). The third argument is the expected sequence number. Note: This function is pure virtual and, therefore, its implementation must be provided by a derived class.This function returns no value and throws no exceptions.

`maxDatagramSize` This method returns the maximum size (in bytes) of any expected, assembled datagram. This value was passed as an argument to the constructor. This function has no arguments and throws no exceptions.

A.3 LCI Assembler

This class allows for the assembly of datagrams which originated from FSW's LAT Charge Injection (LCI) system (see [25]). The implementation of this class is straightforward:

— it inherits from `Assembler` (see Section A.2)

— determines and sets the maximum size of any expected datagrams

— determines and registers the `APID`s corresponding to legitimate LCI datagrams

Note: Because the `APID`s produced by the LCI system are both fixed and determined by FSW, the user calls the base class's `handle` method (see Section 5.2) at their own peril.

The definition for this class is contained in Listing A.2:

Listing A.2 Class definition for `LciAssembler`

```
1:  class LciAssembler : public Assembler {
2:    public:
3:      LciAssembler();
4:    public:
5:      virtual ~LciAssembler();
6:    };
```

A.3.3 Constructor synopsis

Just the default.

A.3.4 Member synopsis

See the base class (Section A.2)

A.4 LPA Assembler

This class allows for the assembly of datagrams which originated from FSW's LAT Physics Analysis (LPA) system (see [25]). The implementation of this class is straightforward:

— it inherits from `Assembler` (see Section A.2)

— determines and sets the maximum size of any expected datagrams

— determines and registers the APIDs corresponding to legitimate LPA datagrams

Note: Because the APIDs produced by the LPA system are both fixed and determined by FSW, the user calls the base class's handle method (see Section 5.2) at their own peril.

The definition for this class is contained in Listing A.3:

Listing A.3 Class definition for LpaAssembler

```
1:  class LpaAssembler : public Assembler {
2:    public:
3:      LpaAssembler();
4:    public:
5:      virtual ~LpaAssembler();
6:    };
```

A.4.5 Constructor synopsis

Just the default.

A.4.6 Member synopsis

See the base class (Section A.2)

A.5 Exceptions

none.

Appendix B
Telemetry from the Monitoring System

The VSC monitors, once per second, 102 analog voltage, current, and temperatures from the LAT and its BPU. Once acquired, this information is transmitted from VSC to proxy as a series of CCSDS packets on the VSC telemetry stream (see Section 7.2). Physically, the monitored information originates from two different types of VSC boards: the *850* and *468* board (see Chapter 2). For each board, the monitoring system generates a single packet, with the packet's APID used to differentiate board. In short, 102 points are monitored and their information is spread over two type of packets giving a transmission rate of two packets per second. This appendix describes, for each type of packet, the telemetry produced by the VSC's monitoring system. These CCSDS packets follow the standard GLAST telemetry standards as described in Section 4.7. Section B.2 describes the telemetry packet produced by the 850 board and Section B.2 describes the telemetry packet produced by the 468 board. In actuality, to mitigate against single point failure, data for these 102 quantities are brought *twice* into the VSC, however, only one set is transmitted at any one time (see Section 3.3.6). Each packet contains, as its last field, a boolean which identifies whether this data originates from the *primary* or *redundant* set of signals. A value of one (1) indicates the primary signals and a value of two (2) indicates the redundant signals. For each quantity monitored, the corresponding VSC digitizer produces 11 bits of magnitude and one bit of *sign*. This 12 bit value is *sign-extended* to a 16-bit value before transmission.

B.1 Unit conversion

There are six different types of conversion necessary to translate between raw (sign extended) ADC counts and engineering units. The relationship between conversion type and monitored quantity is enumerated in the spreadsheets of figures B.1 and B.2. The following six sections define, for each of these six conversion types, the necessary conversion equation.

B.1.1 LAT voltage (LatV)

Equation (1) converts voltage sensor counts (v_n) to voltage (V), where: v_n is the *signed* 16-bit ADC value and V is measured in *volts*.

$$\begin{aligned} V &= c_0 / c_1 \times v_n \\ &\text{where:} \\ c_0 &= 20\,V \\ c_1 &= 4096\,counts \end{aligned} \tag{1}$$

B.1.2 BPU voltage (BpuV)

Equation (2) converts BPU voltage sensor counts (v_n) to voltage (V), where: v_n is the *signed* 16-bit ADC value and V is measured in *volts*.

$$\begin{aligned} V &= c_0 / c_1 \times v_n \\ &\text{where:} \\ c_0 &= 200\,V \\ c_1 &= 4096\,counts \end{aligned} \tag{2}$$

B.1.3 BPU Current (Bpul)

Equation (3) converts BPU current sensor counts (v_n) to current (A), where: v_n is the *signed* 16-bit ADC value and A is measured in *amperes*.

$$\begin{aligned} A &= c_0 / c_1 \times v_n \\ &\text{where:} \\ c_0 &= 20\,A \\ c_1 &= 4096\,counts \end{aligned} \tag{3}$$

B.1.4 DAQ Current (Daql)

Equation (4) converts DAQ current sensor counts (v_n) to current (A), where: v_n is the *signed* 16-bit ADC value and A is measured in *amperes*.

$$
\begin{aligned}
&A = c_0/c_1 \times v_n \\
&\text{where:} \\
&c_0 = 200\,\mathrm{A} \\
&c_1 = 4096\,\mathrm{counts}
\end{aligned}
\tag{4}
$$

B.1.5 Thermistor

Equation (5) converts thermistor sensor counts (v_n) to resistance (r), where: v_n is the *signed* 16-bit ADC value and r is measured in *kilo-ohms*. The conversion from resistance to temperature is specified in reference [29].

$$
\begin{aligned}
&r = c_0/v_n - c_1 \\
&\text{where:} \\
&c_0 = 191488\,\mathrm{k\Omega counts} \\
&c_1 = 120\,\mathrm{k\Omega}
\end{aligned}
\tag{5}
$$

B.1.6 RTD

Equation (6) converts RTD sensor counts (v_n) to resistance (r), where: v_n is the *signed* 16-bit ADC value and r is measured in *kilo-ohms*. The conversion from resistance to temperature is specified in reference [28].

$$
\begin{aligned}
&r = (c_0 + c_1 \times v_n)/(c_2 - c_3 \times v_n) \\
&\text{where:} \\
&c_0 = 775.5\,\mathrm{k\Omega} \\
&c_1 = 0.097656\,\mathrm{k\Omega/count} \\
&c_2 = 361.25 \\
&c_3 = 0.049142/\mathrm{count}
\end{aligned}
\tag{6}
$$

B.2 Telemetry from the 850 board

The spread-sheet illustrated in Figure B.1 enumerates the quantities monitored by the *850* board and encapsulated in the telemetry packet whose APID is defined in Section B.2.7. Each entry in the spread-sheet corresponds to one, specific monitored quantity. For each quantity, the spread-sheet specifies the relative location of its data, sensor type, and description. Data offsets are described in units of 16-bit (decimal) words with offset *zero* (0) beginning immediately following the standard telemetry header (i. e., the start of the packet's *payload*).

For example, offset eight (8) corresponds to a measurement of the GASU's temperature on its primary DAQ board. The sensor type is "thermistor" which specifies temperature as measured through a thermistor.

B.2.7 APID = 0x00A8

Offset	Conversion	Description
0	Thermistor	Primary SIU Temperature
1	LatV	Primary SIU Voltage
2	Thermistor	Primary SIU Spare Temperature
3	LatV	Primary SIU Spare Voltage
4	Thermistor	Redundant SIU Temperature
5	LatV	Redundant SIU Voltage
6	Thermistor	Redundant SIU Spare Temperature
7	LatV	Redundant SIU Spare Voltage
8	Thermistor	Primary GASU DAQ Board Temperature
9	LatV	Primary GASU DAQ Board Converter AEM/EBM Digital Voltage (3.3V)
10	LatV	Primary GASU DAQ Board Converter AEM/EBM Digital Voltage (2.5V)
11	LatV	Primary GASU DAQ Board Converter CRU/GEM Digital Voltage (3.3V)
12	LatV	Primary GASU DAQ Board Converter CRU/GEM Digital Voltage (2.5V)
13	Thermistor	Primary GASU DAQ Board Spare Temperature
14	LatV	Primary GASU DAQ Board Spare Voltage 1
15	LatV	Primary GASU DAQ Board Spare Voltage 2
16	Thermistor	Redundant GASU DAQ Board Temperature
17	LatV	Redundant GASU DAQ Board Converter AEM/EBM Digital Voltage (3.3V)
18	LatV	Redundant GASU DAQ Board Converter AEM/EBM Digital Voltage (2.5V)
19	LatV	Redundant GASU DAQ Board Converter CRU/GEM Digital Voltage (3.3V)
20	LatV	Redundant GASU DAQ Board Converter CRU/GEM Digital Voltage (2.5V)
21	Thermistor	Redundant GASU DAQ Board Spare Temperature
22	LatV	Redundant GASU DAQ Board Spare Voltage 1
23	LatV	Redundant GASU DAQ Board Spare Voltage 2
24	Thermistor	PDU 0 Board Temperature
25	LatV	PDU 0 Voltage
26	Thermistor	PDU 1 Board Temperature
27	LatV	PDU 1 Voltage
28	LatV	PDU Spare Voltage 1
29	LatV	PDU Spare Voltage 2
30	LatV	Analog sum of primary VCHP heater switch outputs on the +Y radiator
31	LatV	Analog sum of primary VCHP heater switch outputs on the -Y radiator
32	LatV	Analog sum of redundant VCHP heater switch outputs on the +Y radiator
33	LatV	Analog sum of redundant VCHP heater switch outputs on the -Y radiator
34	LatV	+Y Heater Control Box Spare Voltage 1
35	LatV	+Y Heater Control Box Spare Voltage 2
36	LatV	-Y Heater Control Box Spare Voltage 1
37	LatV	-Y Heater Control Box Spare Voltage 2
38	BpuV	Regulated BPU SIU feed Voltage
39	BpuI	Regulated BPU SIU feed Current
40	BpuV	Regulated BPU DAQ feed Voltage
41	DaqI	Regulated BPU DAQ feed Current
42	BpuV	Regulated BPU VCHP Voltage
43	BpuI	Regulated BPU VCHP Current
44	BpuV	Unregulated BPU Primary Grid Radiator Voltage
45	BpuI	Unregulated BPU Primary Grid Radiator Current
46	BpuV	Unregulated BPU Redundant Grid Radiator Voltage
47	BpuI	Unregulated BPU Redundant Grid Radiator Current
48	NA	Which SC monitor was used to generate this packet

Figure B.1 Enumeration of 850 board monitored quantities

B.3 Telemetry from the 468 board

The spread-sheet illustrated in Figure B.2 enumerates the quantities monitored by the *468* board and encapsulated in the telemetry packet whose APID is defined in Section B.3.8. Each entry in the spread-sheet corresponds to one, specific monitored quantity. For each quantity, the spread-sheet specifies the relative location of its data, sensor type, and description. Data offsets are described in units of 16-bit (decimal) words with offset *zero* (0) beginning immediately following the standard telemetry header (i. e., the start of the packet's *payload*). For example, offset twenty-six (26) corresponds to a measurement of the $+X$ tile temperature. The sensor type is "Rtd" and specifies temperature as measured through a RTD.

B.3.8 APID = 0x00A4

Offset	Conversion	Description
0	Rtd	+Y Radiator Panel Upper Left Survival Heater Temperature
1	Rtd	+Y Radiator Panel Upper Right Survival Heater Temperature
2	Rtd	+Y Radiator Panel Lower Left Survival Heater Temperature
3	Rtd	+Y Radiator Panel Lower Right Survival Heater Temperature
4	Rtd	+Y VCHP Reservoir Heater Temperature 0
5	Rtd	+Y VCHP Reservoir Heater Temperature 1
6	Rtd	+Y VCHP Reservoir Heater Temperature 2
7	Rtd	+Y VCHP Reservoir Heater Temperature 3
8	Rtd	+Y VCHP Reservoir Heater Temperature 4
9	Rtd	+Y VCHP Reservoir Heater Temperature 5
10	Thermistor	+Y VCHP-XLHP Interface +X Side Temperature
11	Thermistor	+Y VCHP-XLHP Interface -X Side Temperature
12	Thermistor	+Y Spare Temperature channel 0
13	Rtd	-Y Radiator Panel Upper Left Survival Heater Temperature
14	Rtd	-Y Radiator Panel Upper Right Survival Heater Temperature
15	Rtd	-Y Radiator Panel Lower Left Survival Heater Temperature
16	Rtd	-Y Radiator Panel Lower Right Survival Heater Temperature
17	Rtd	-Y VCHP Reservoir Heater Temperature 0
18	Rtd	-Y VCHP Reservoir Heater Temperature 1
19	Rtd	-Y VCHP Reservoir Heater Temperature 2
20	Rtd	-Y VCHP Reservoir Heater Temperature 3
21	Rtd	-Y VCHP Reservoir Heater Temperature 4
22	Rtd	-Y VCHP Reservoir Heater Temperature 5
23	Thermistor	-Y VCHP-XLHP Interface -X Side Temperature
24	Thermistor	-Y VCHP-XLHP Interface +X Side Temperature
25	Thermistor	-Y Spare Temperature channel 0
26	Rtd	+X ACD Tile Temperature
27	Rtd	-X ACD Tile Temperature
28	Rtd	+Y ACD Tile Temperature
29	Rtd	-Y ACD Tile Temperature
30	Rtd	+Z ACD Tile Temperature
31	Thermistor	+X ACD Inside Composite Shell Temperature
32	Thermistor	-X ACD Inside Composite Shell Temperature
33	Thermistor	+Y ACD Inside Composite Shell Temperature
34	Thermistor	-Y ACD Inside Composite Shell Temperature
35	Thermistor	+Z ACD Inside Composite Shell Temperature
36	Thermistor	-X ACD PMT Rail Right Temperature
37	Thermistor	-Y ACD PMT Rail Right Temperature
38	Thermistor	+X ACD PMT Rail Right Temperature
39	Thermistor	+Y ACD PMT Rail Right Temperature
40	Thermistor	X-LAT Plate Heat Pipe Temperature 1
41	Thermistor	X-LAT Plate Heat Pipe Temperature 2
42	Thermistor	X-LAT Plate Heat Pipe Temperature 3
43	Thermistor	X-LAT Plate Heat Pipe Temperature 4
44	Thermistor	+Y Grid-Radiator Interface +X Side Temperature
45	Thermistor	+Y Grid-Radiator Interface -X Side Temperature
46	Thermistor	+Y Grid Make-up Heaters +X Side Temperature
47	Thermistor	+Y Grid Make-up Heaters -X Side Temperature
48	Thermistor	+Y Spare Temperature channel 1
49	Thermistor	-Y Grid-Radiator Interface -X Side Temperature
50	Thermistor	-Y Grid-Radiator Interface +X Side Temperature
51	Thermistor	-Y Grid Make-up Heaters -X Side Temperature
52	Thermistor	-Y Grid Make-up Heaters +X Side Temperature
53	Thermistor	-Y Spare Temperature channel 1
54	NA	Which SC monitor was used to generate this packet

Figure B.2 Enumeration of 468 board monitored quantities